Documentation de conception

GL28

Janvier 2020

Table des matières

1 Liste des classes et leur dépendances

Cette liste non exhaustive des **classes Java** a pour vocation d'introduire l'architecture globale du compilateur. On y retrouve les diagrammes UML (Unified Modeling Language) de ces classes afin d'identifier rapidement les dépendances ainsi que les relation de hiérarchie.

1.1 Compilateur et options

1.1.1 DecacMain

Point d'entrée de la commande *decac*, réalise un appel à **CompilerOptions** pour gérer les options et **DecacCompiler** pour effectuer la compilation.

1.1.2 CompilerOptions

Gestion des options - une amélioration possible l'implémentation de l'option *-P* : parallélisme de la compilation de plusieurs programmes Deca.

1.1.3 DecacCompiler

Exécute les différentes étapes de la compilation dans la méthode doCompile : un noeud **Program** est généré après l'action du lexeur et parseur, puis on appelle la méthode *verifyProgram* pour effectuer les vérifications et décorations contextuelles. Enfin *display* fournit un code que le programme *ima* est capable de lire.

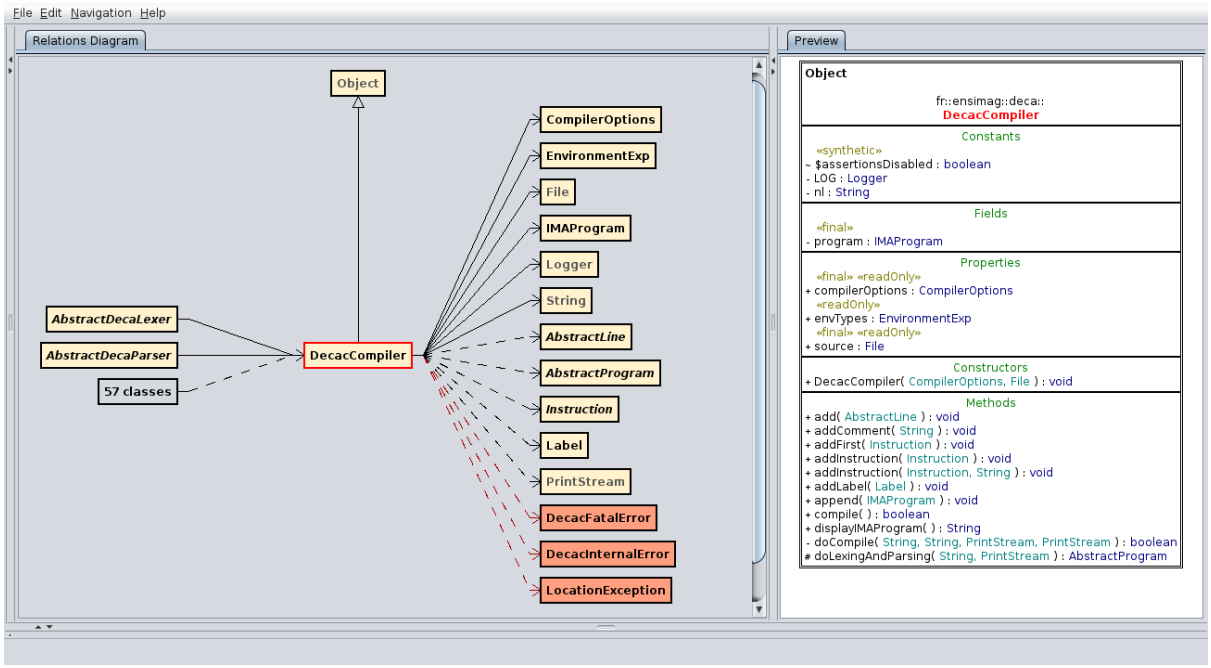

FIGURE 1 – UML diagram of **DecacCompiler**

1.2 Définitions et types

Afin de décorer l'arbre de syntaxe abstraite, on utilise respectivement les définitions pour les identifiants (qui représentent les variables, méthodes, champs, etc ..) et les types pour les expressions. Nous présentons brièvement ci dessous leurs sous classes et spécificités.

1.2.1 Definition

Une définition est la donnée d'un **Type** et d'une **Location**.

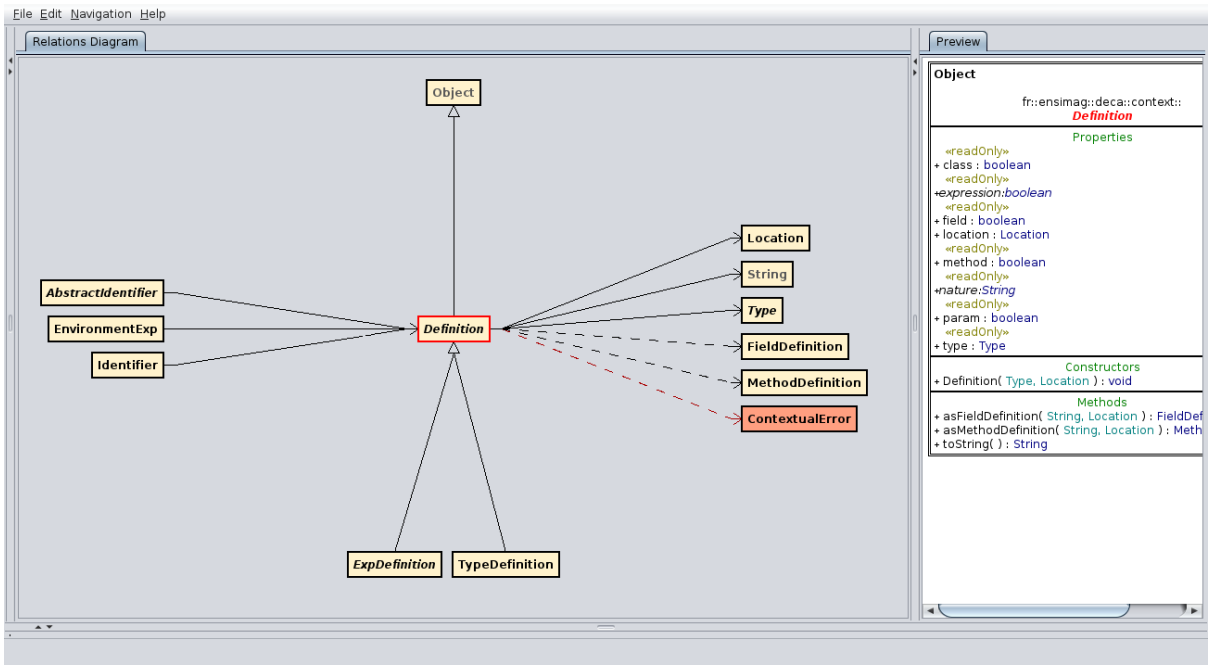

FIGURE 2 – UML diagram of **Definition**

1.2.2 ExpDefinition

Classe parent des définitions associées aux champs, méthodes, paramètres et variables.

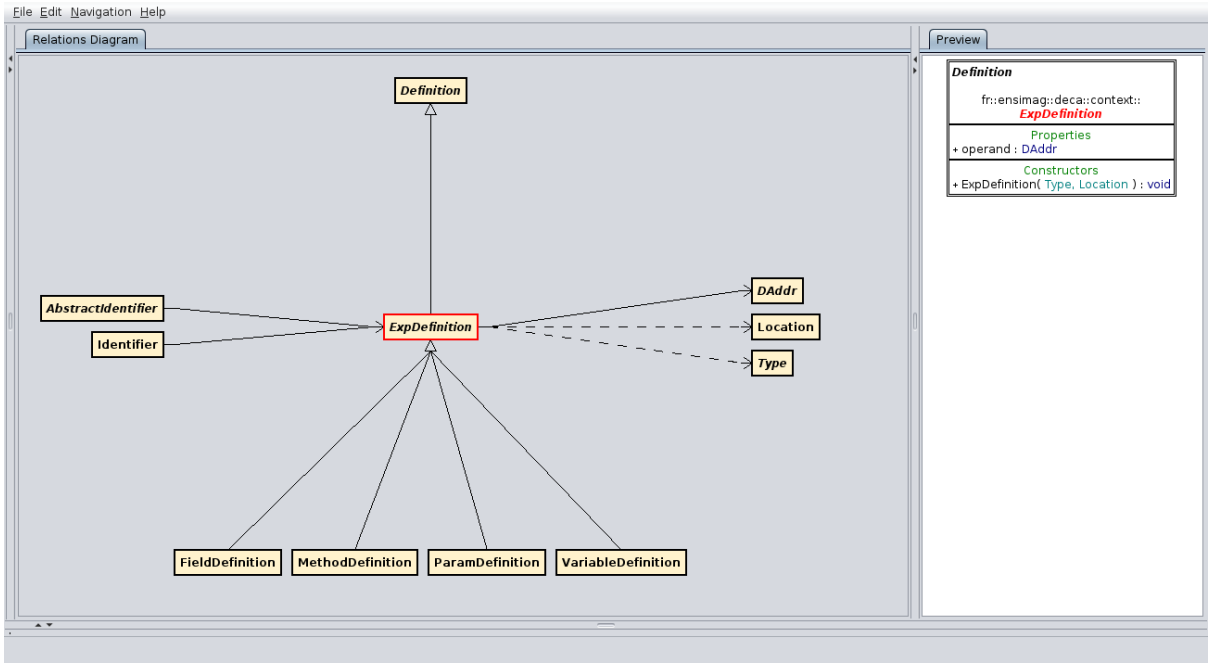

FIGURE 3 – UML diagram of **ExpDefinition**

1.2.3 VariableDefinition

La définition d'une variable est la donnée d'un **Type**, d'une **Location**.

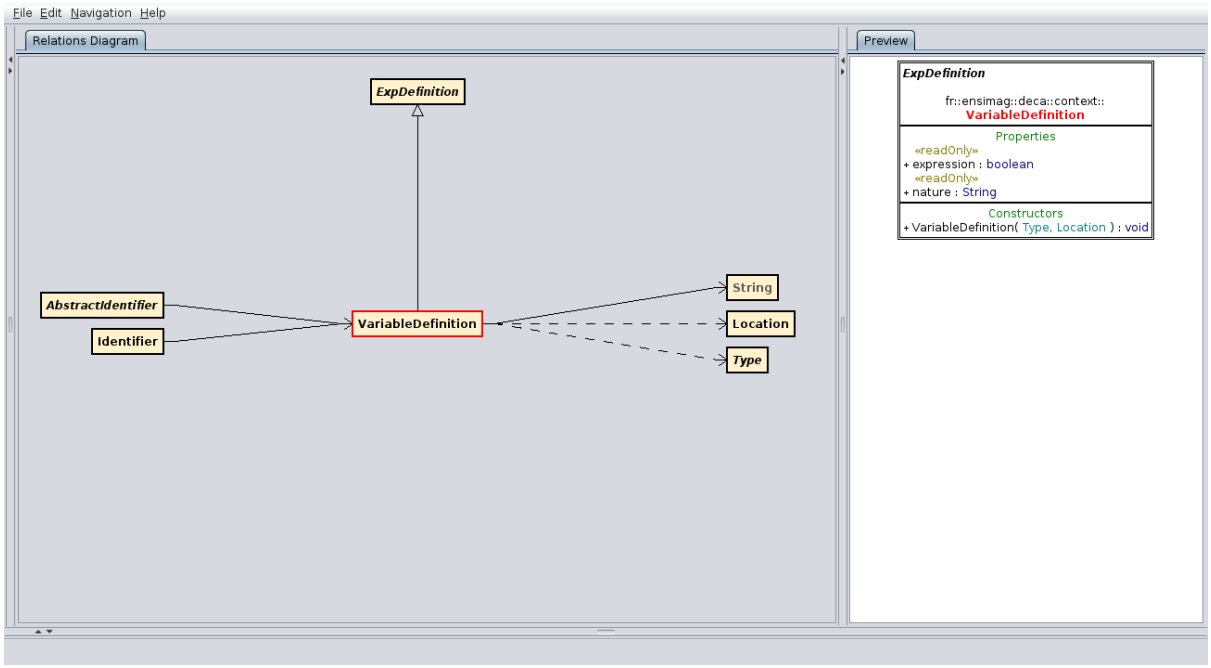

FIGURE 4 – UML diagram of **VariableDefinition**

1.2.4 MethodDefinition

La définition d'une méthode est la donnée d'un **Type** (le type de retour), d'une **Location**, d'une **Signature** et d'un entier correspond à la position de la méthode dans l'ensemble de ses super-classes. Une signature est implémentée comme une *List*<**Type**> et correspond donc aux types des paramètres de la méthode.

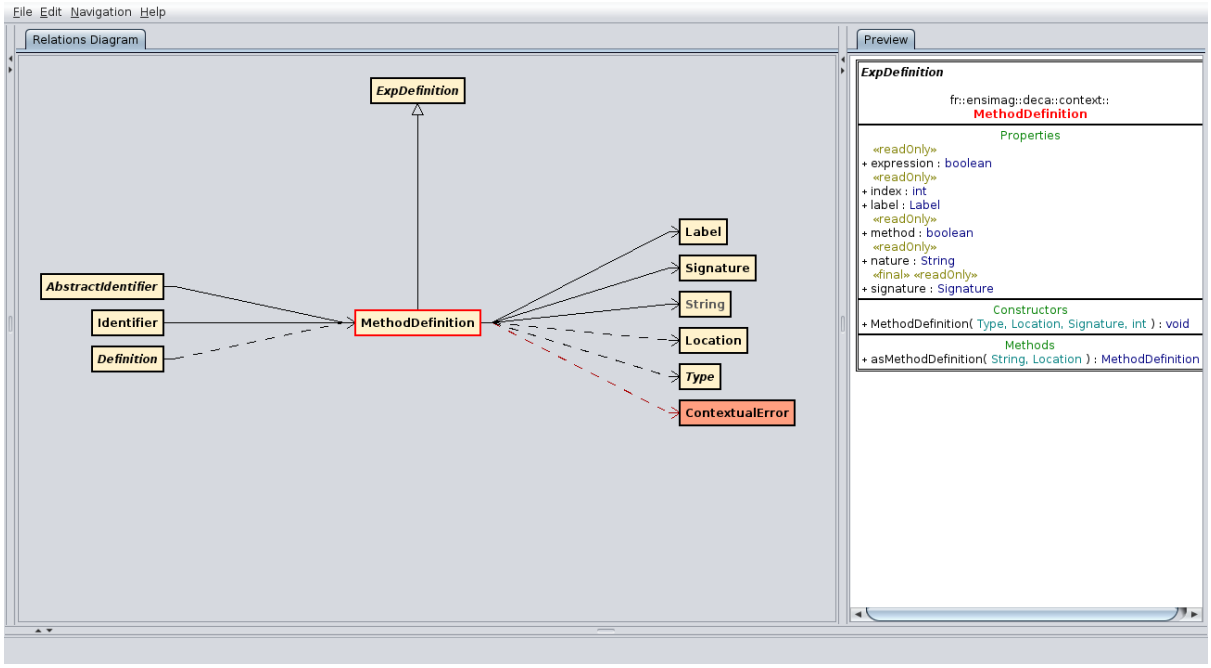

FIGURE 5 – UML diagram of **MethodDefinition**

1.2.5 FieldDefinition

La définition d'un champ est la donnée d'un **Type**, d'une **Location**, d'une **Visibility** (*public* ou *protected*), de la classe dont il est membre, et d'un entier correspond à la position du champ dans l'ensemble de ses super-classes.

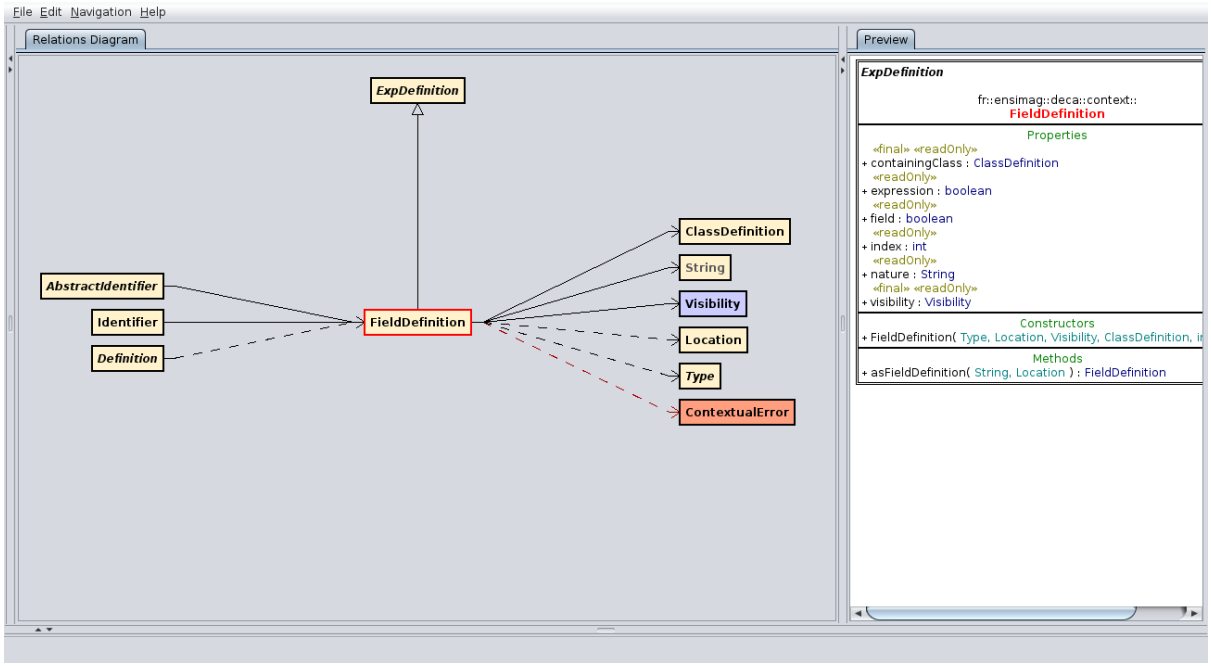

FIGURE 6 – UML diagram of **FieldDefinition**

1.2.6 ParamDefinition

La définition d'une variable est la donnée d'un **Type**, d'une **Location**.

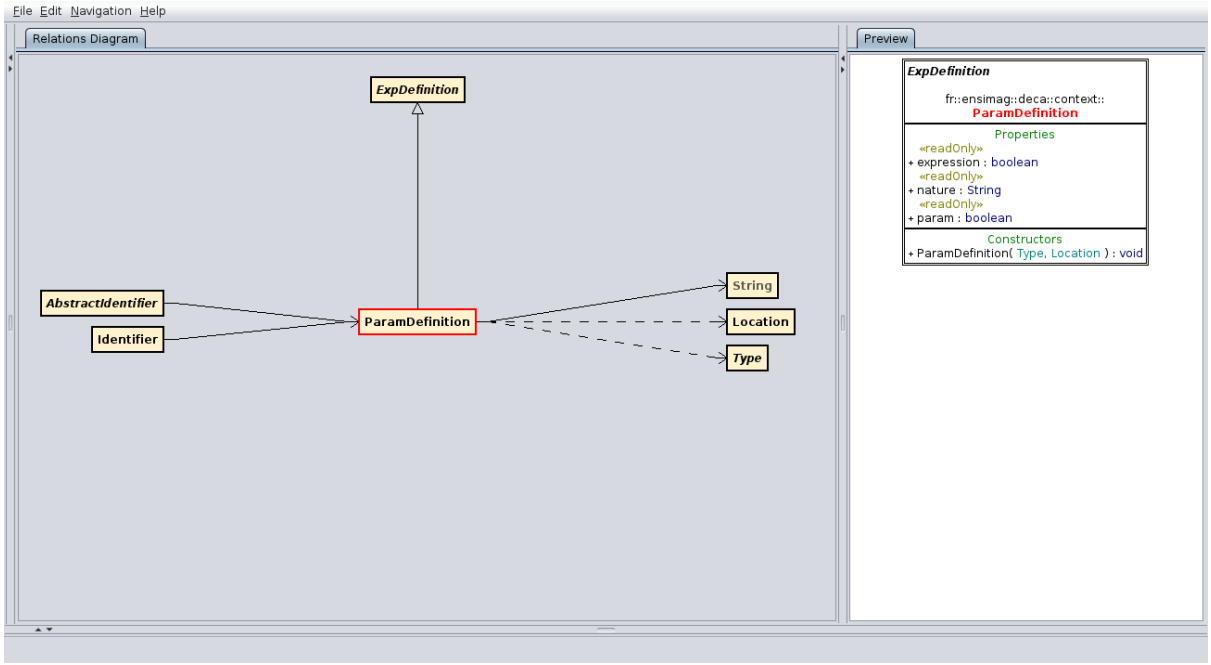

FIGURE 7 – UML diagram of **ParamDefinition**

1.2.7 TypeDefinition

Un type correspond à un type prédéfini (cf sous classes de **Type**) ou à une classe

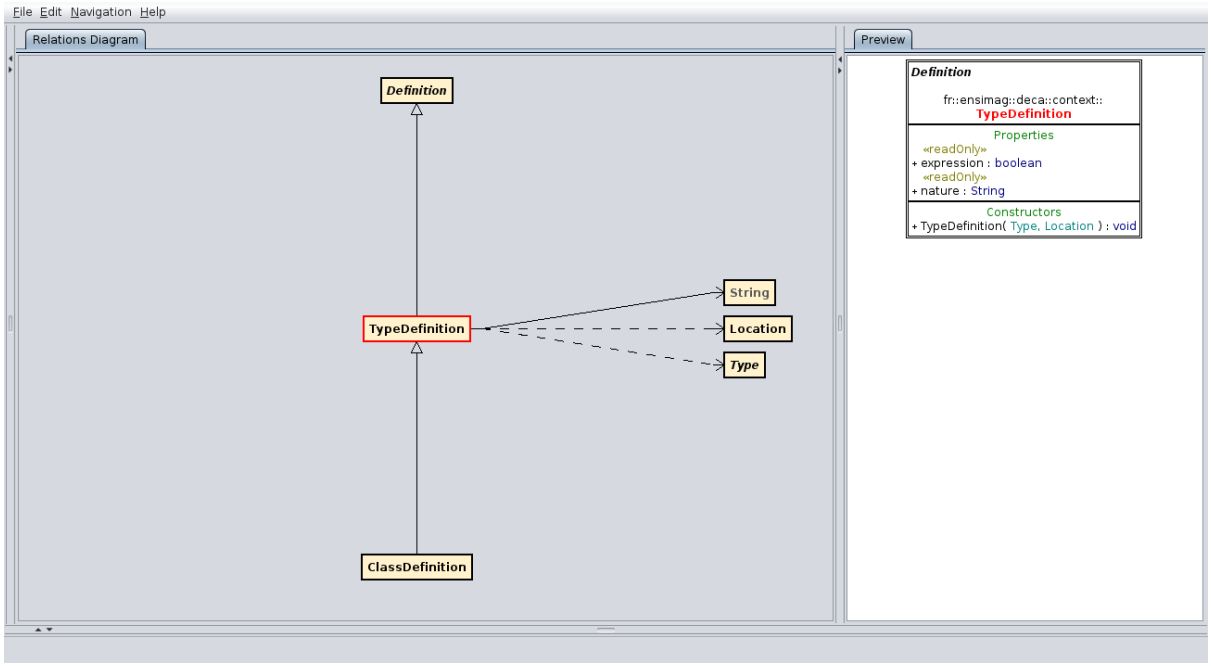

FIGURE 8 – UML diagram of **TypeDefinition**

1.2.8 ClassDefinition

Ses champs contiennent un **EnvironmentExp** des champs et méthodes de la classe, leur nombre respectif ainsi que sa super-classe.

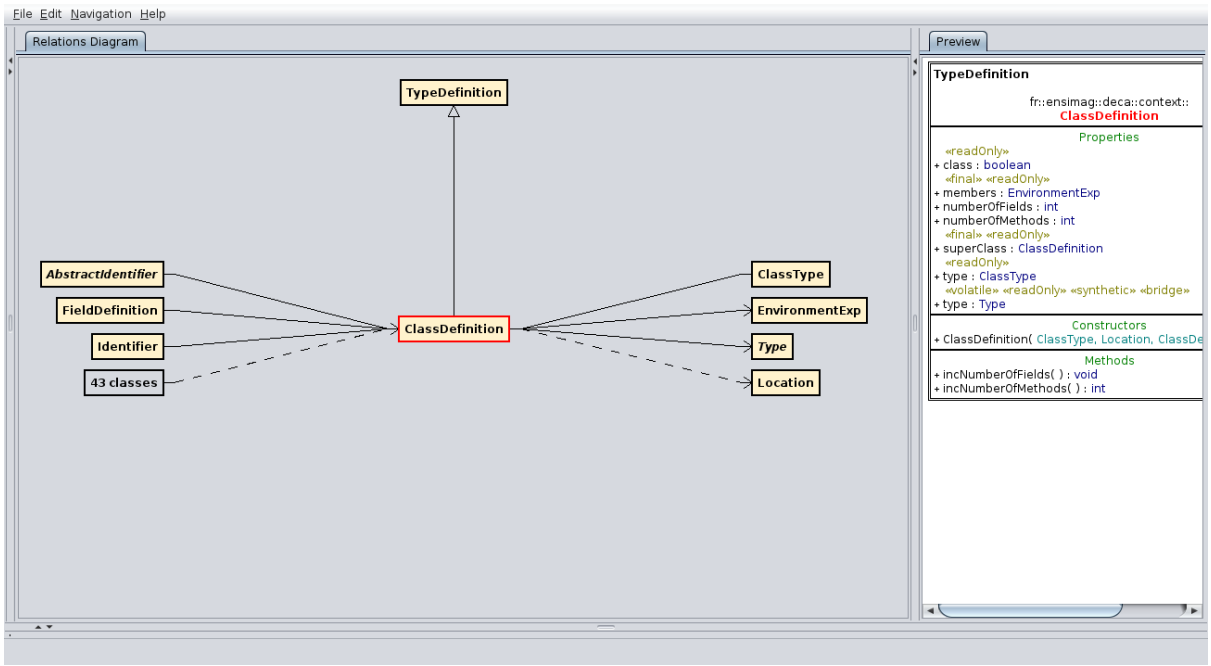

FIGURE 9 – UML diagram of **ClassDefinition**

1.2.9 ClassType

On utilise ce type afin d'ajouter une nouvelle classe dans l'environnement des types.

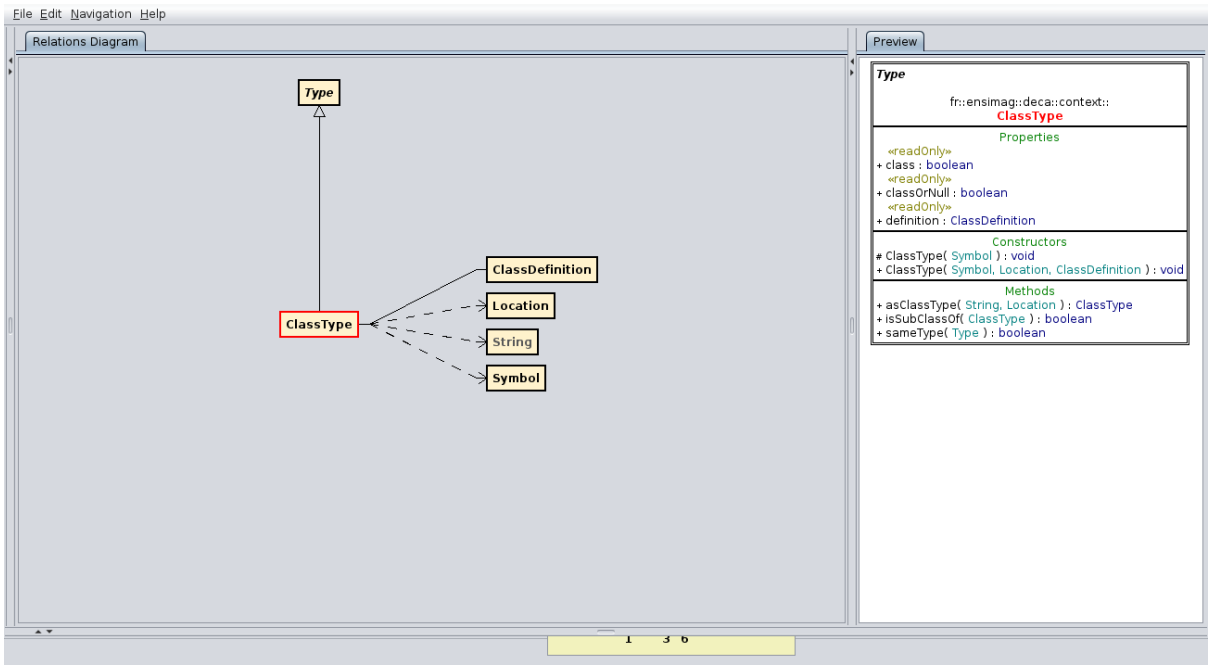

FIGURE 10 – UML diagram of **ClassType**

1.2.10 Type

Les sous classes de **Type** définissent les types prédéfinis du langage Deca. Ils sont ajoutés à l'environnement des types lors la création de ce dernier au lancement du compilateur, en tant que champ de **DecacCompiler**. Cet environnement est hérité dans la totalité des vérifications contextuelles.

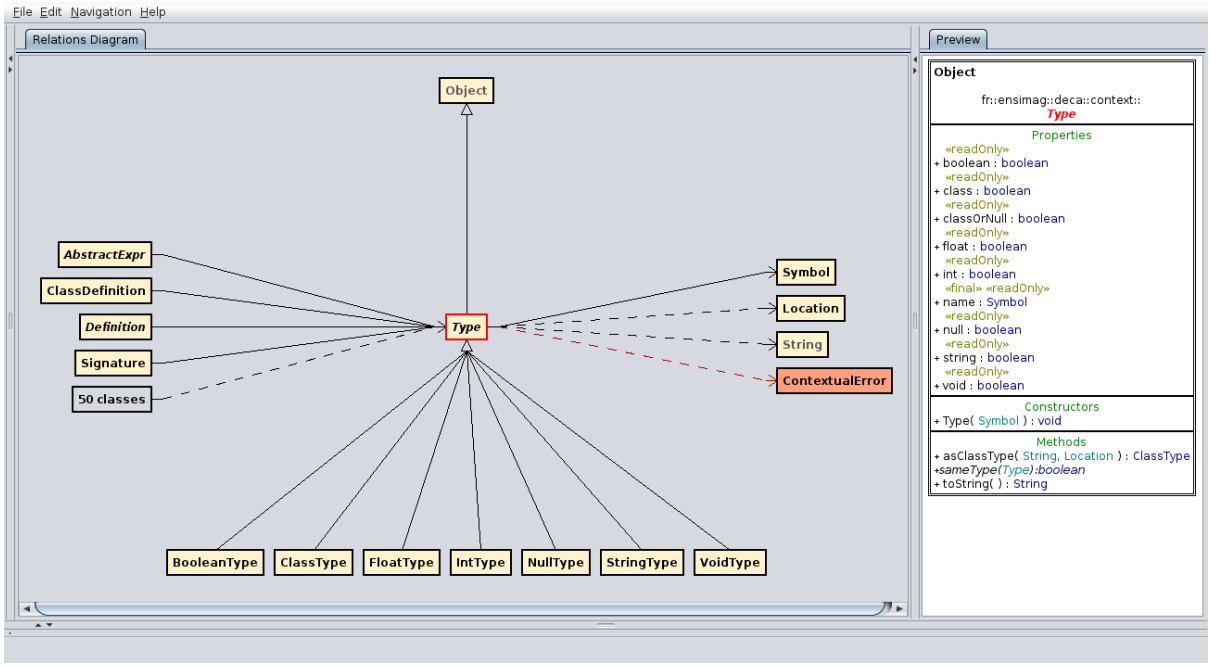

FIGURE 11 – UML diagram of **Type**

1.2.11 EnvironmentExp

Implémente les structures de données associées aux identifiants sous la forme d'une table de hachage *Map*<**Symbol**, **Definition**>. En outre, afin de traiter la hiérarchie des **ExpDefinition** dans un programme Deca, cette classe dispose d'un champ associé à la table de hachage *parent*. En pratique, on peut ainsi appeler la méthode *get* pour chercher une définition dans la table "courante", et la méthode *getAny* pour remonter le chaînage des tables tant que la clé ne correspond à aucune définition. Pour vérifier si un type correspond à un type prédéfini, on utilise *getDefinitionFromName*.

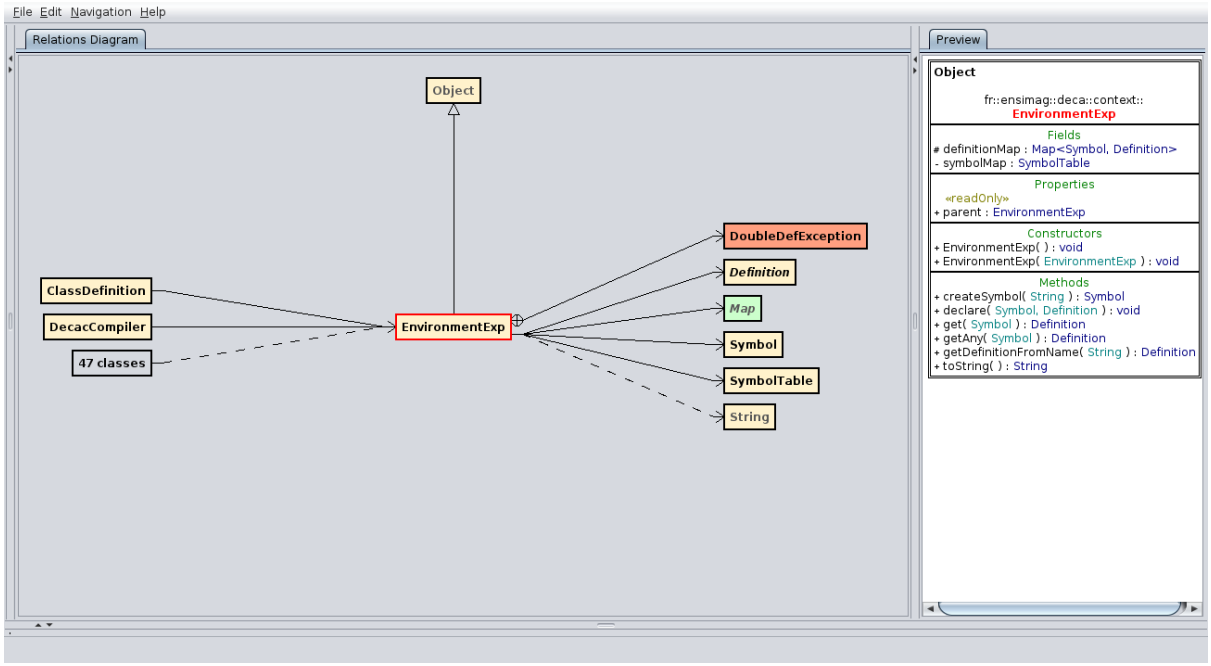

FIGURE 12 – UML diagram of **EnvironmentExp**

1.3 Éléments de l'arbre de syntaxe abstraite

1.3.1 Tree

Toutes les classes suivantes ont en commun une classe parent **Tree** qui implémente ou définit des méthodes utiles à la décompilation (*decompile(..)*) et au débogage de l'arbre généré en sortie d'étape A et sa version décorée en sortie d'étape B (*prettyPrintXyz(..)*).

1.3.2 TreeList

La classe **TreeList** permet de facilement itérer sur des éléments de l'arbre et est implémentée pour de nombreuses classes afin d'obtenir une liste de déclarations ou d'instructions.
Dans ces sous classes **ListXyZ**, on se contente ainsi d'itérer sur les sous classes de **Tree** associées et on appelle les méthodes de vérification contextuelle et de génération de code assembleur de ces sous classes.

1.3.3 AbstractInst

AbstractInst définit un non terminal correspondant à une instruction du langage Deca. De nombreuses classes l'implémentent, comme par exemple le non terminal **AbstractExpr** ou le terminal **Return**. Concrètement, c'est dans ces sous classes que les étapes B et C sont codées.

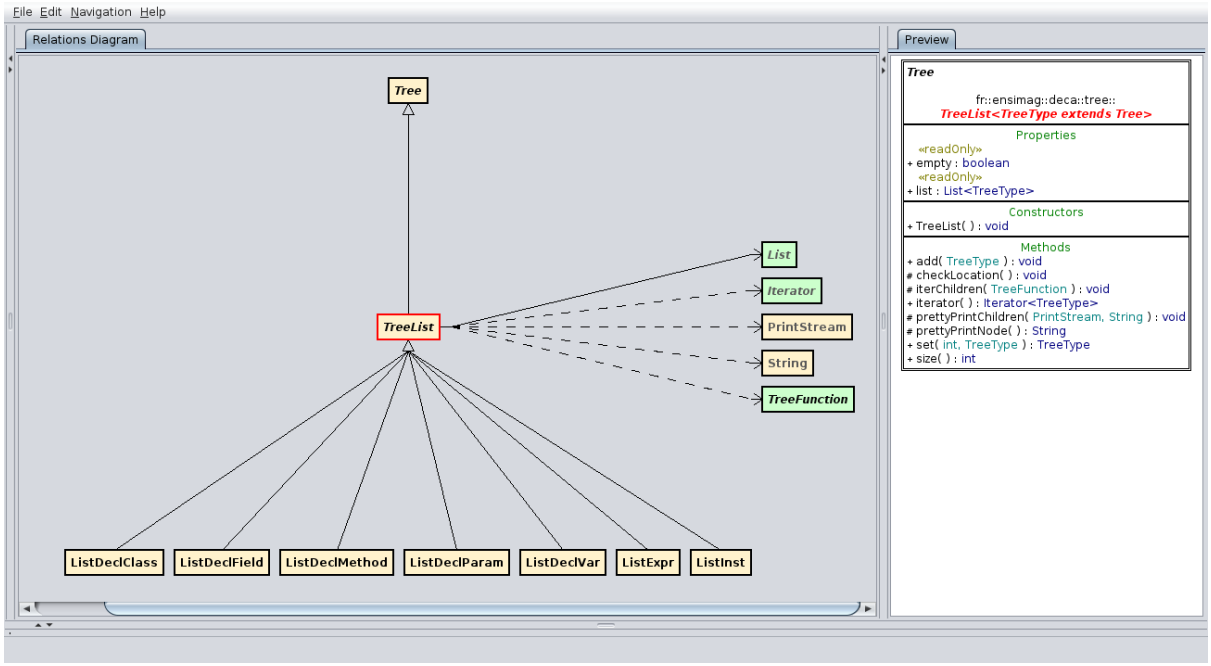

FIGURE 13 – UML diagram of class TreeList

1.3.4 IfThenElse

La classe **IfThenElse** possède trois attributs : une condition **AbstractExpr** dont on vérifie qu'il s'agit d'une expression booléenne, ainsi que de deux **ListInst** des branches *Then* et *Else*.

1.3.5 NoOperation

Le terminal **NoOperation** correspond à une instruction comprenant uniquement un point virgule, il n'y a rien à vérifier ou générer.

1.3.6 Return

Le terminal **Return** correspond à une instruction *return Xyz ;*, on vérifie que le type renvoyé correspondant bien à ce qu'on attendait.

1.3.7 While

La classe **While** est similaire à **IfThenElse**, et comprend une condition et une branche d'instructions.

1.3.8 AbstractPrint

Le non terminal **AbstractPrint** implémente les terminaux d'affichage **Print** et **Println**, on doit vérifier que le type des arguments sont licites, c'est à dire une valeur numérique ou une chaîne de caractères.

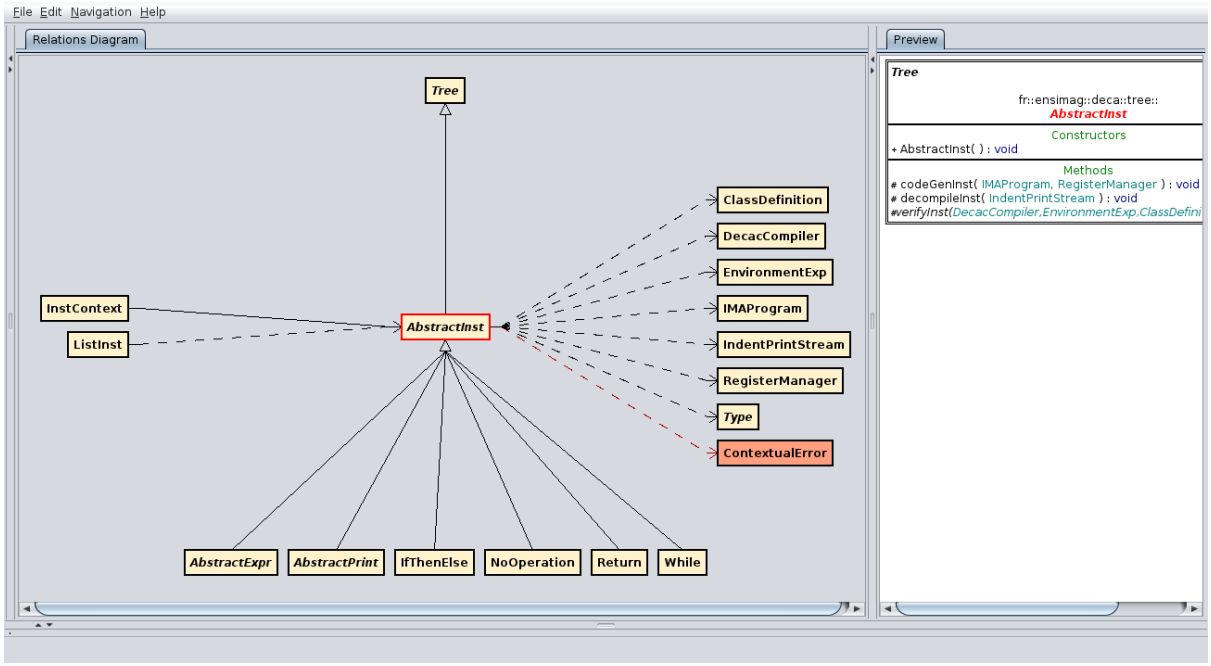

FIGURE 14 – UML diagram of class AbstractInst

1.3.9 AbstracExpr

Cette classe est centrale dans l'étape de syntaxe contextuelle. Elle introduit les méthodes de vérification et décoration *verifyExpr*, *verifyCondition* et *verifyRValue* . Seules verifyCondition et verifyRValue sont implémentées dans cette classe. La méthode *verifyExpr* est implémentée dans les nombreuses sous classes de **AbstractExpr** et consiste principalement à affecter un type aux terminaux qui composent ces expressions.

1.3.10 AbstractBinaryExpr

La classe abstraite **AbstractBinaryExpr** définit les expressions binaires composées de deux opérandes **AbstractExpr** et d'un opérateur.

<

1.3.11 Sous classes de AbstractBinaryExpr

L'ensemble des ces sous classes consistent en la redéfinition de *verifyExpr* afin de respecter leur spécificité comme décris dans la syntaxe contextuelle du langage Deca. Voir ci dessous l'ensemble des opérateurs et leur hiérarchie.

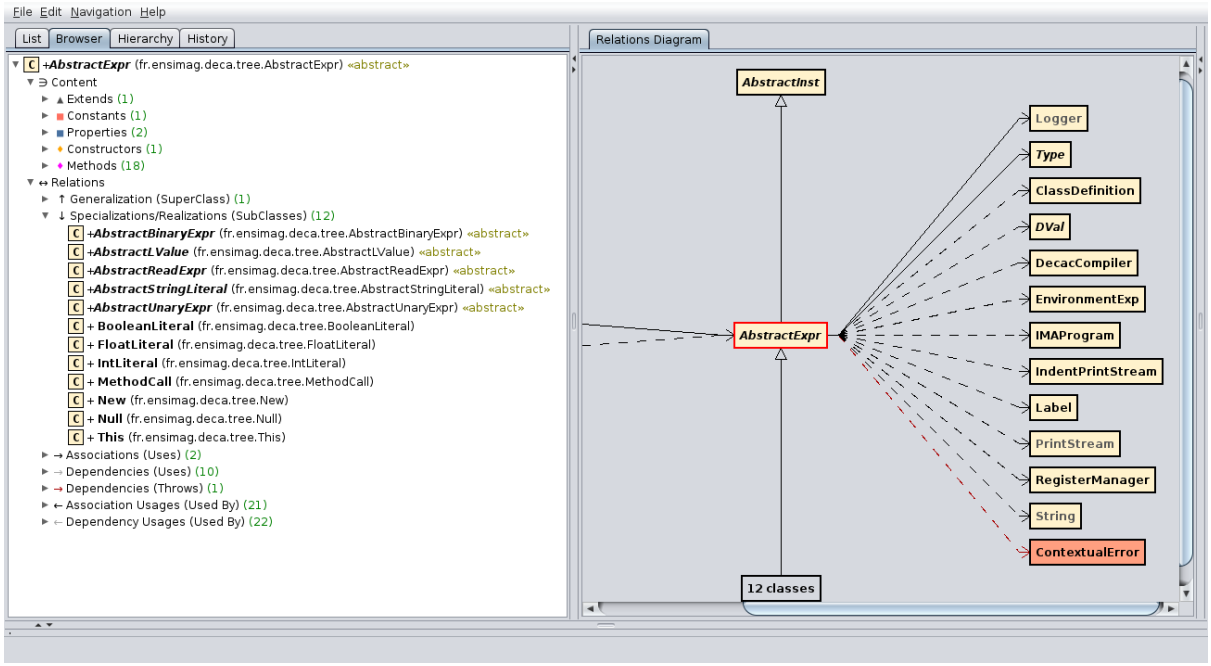

FIGURE 15 – Subclasses(left) and dependencies(right) of **AbstractExpr**

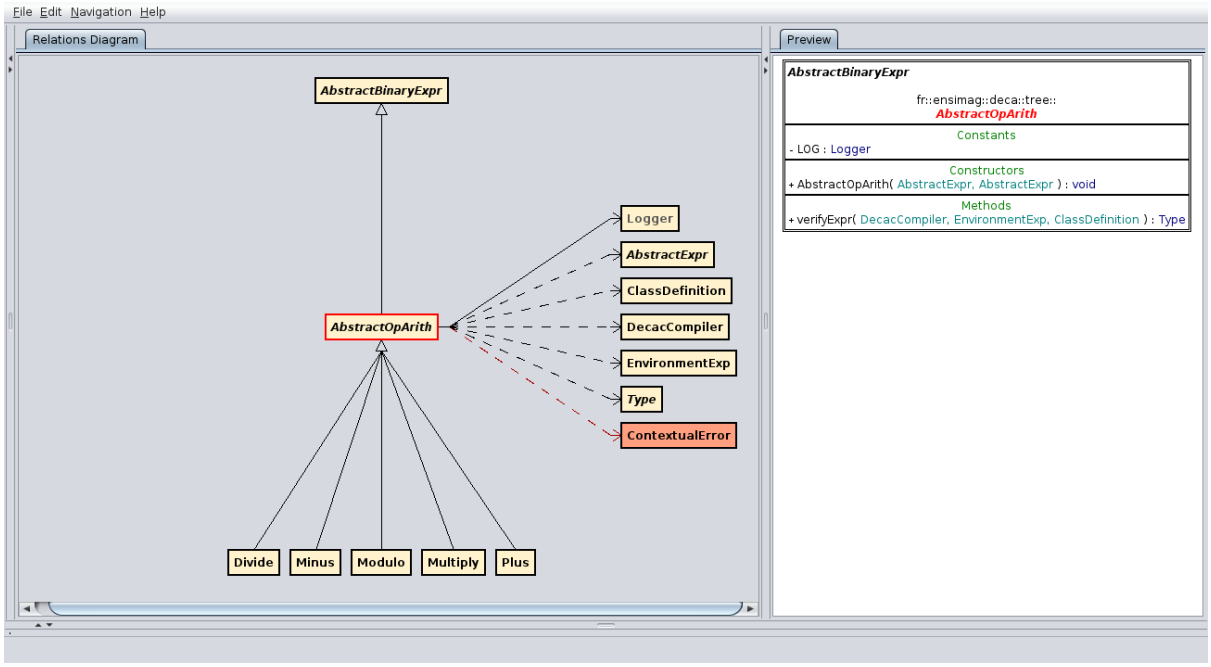

FIGURE 17 – UML diagram of **AbstractOpArith**

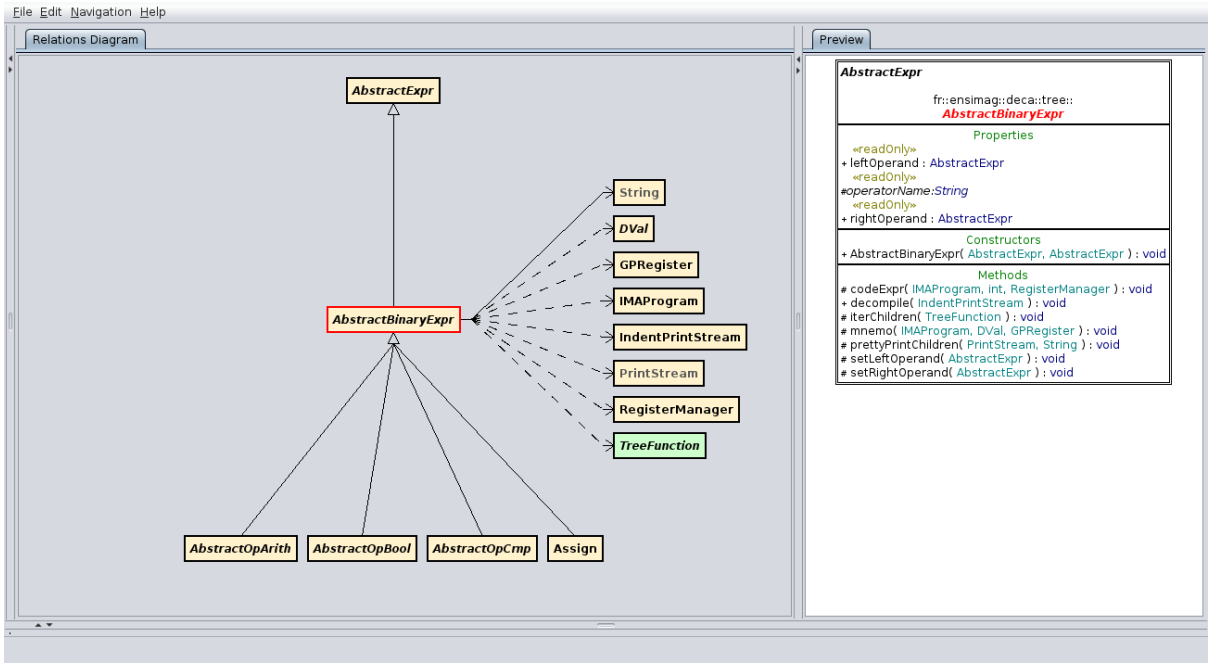

FIGURE 16 – UML diagram of **AbstractBinaryExpr**

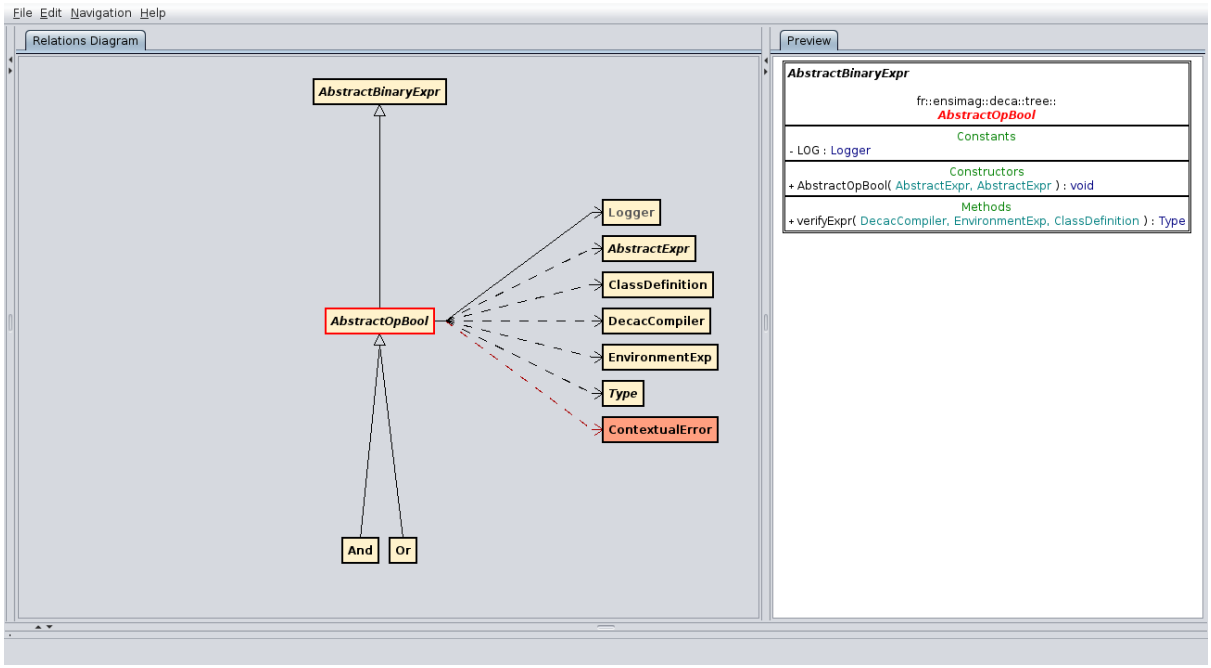

FIGURE 18 – UML diagram of **AbstractOpBool**

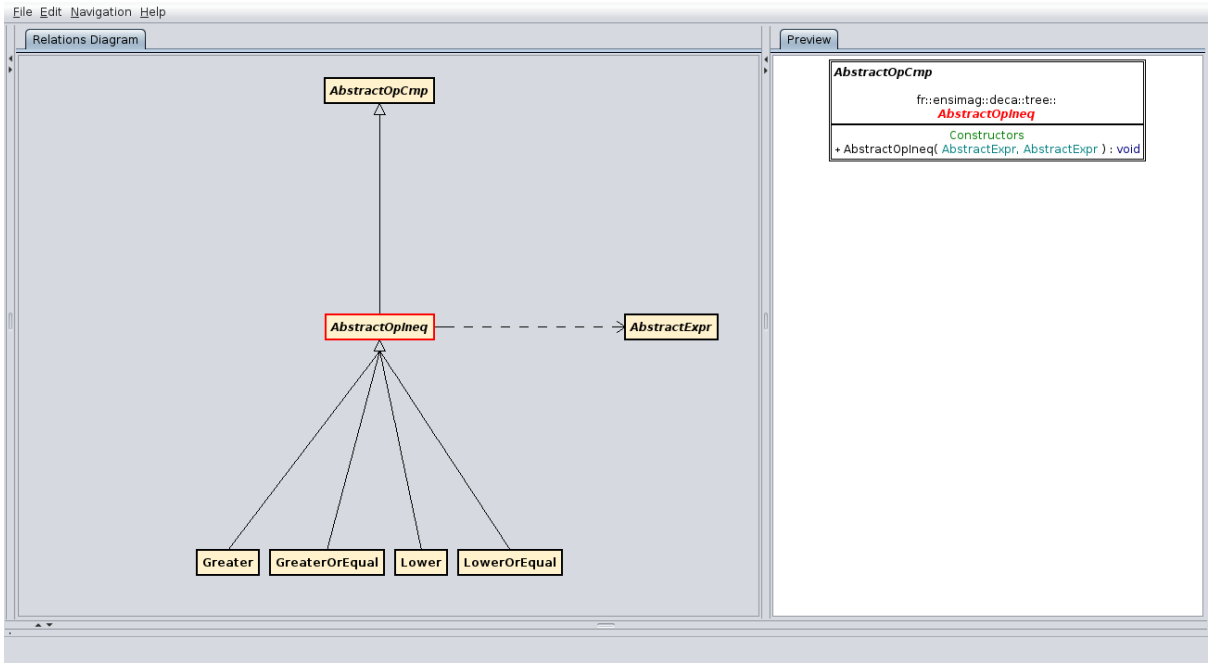

FIGURE 19 – UML diagram of **AbstractOpIneq**

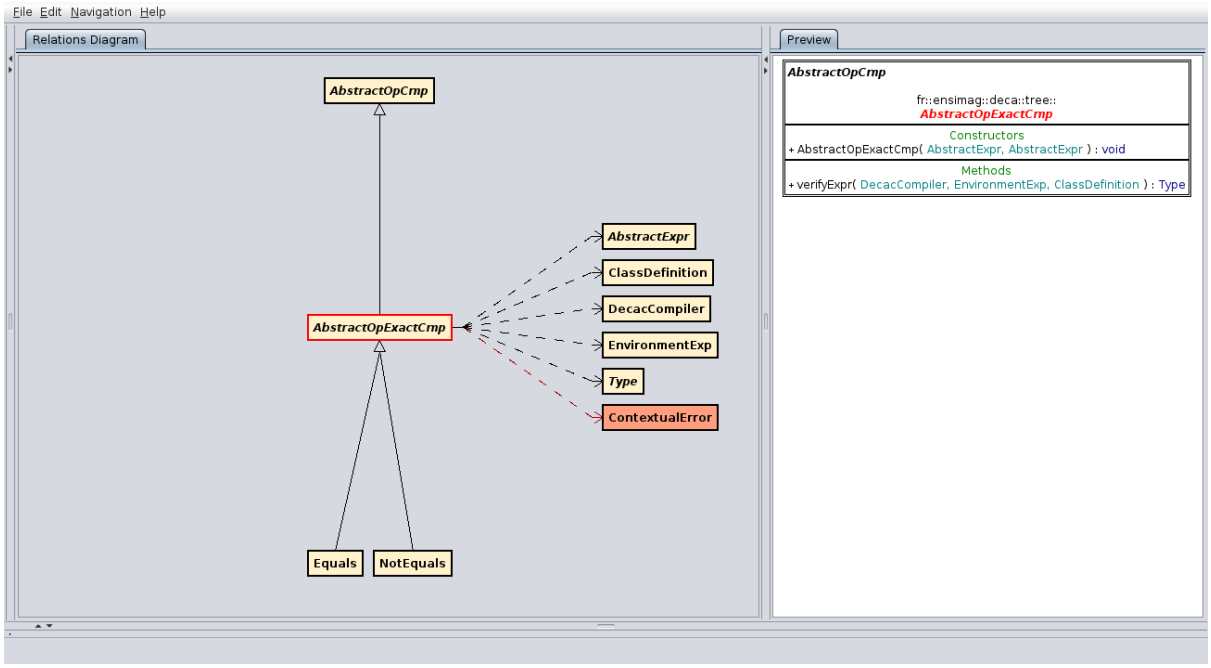

FIGURE 20 – UML diagram of **AbstractOpExactCmp**

1.3.12 AbstractUnaryExpr

Implémentation semblable aux expressions binaires, on y définit les noeuds **Not**, **UnaryMinus** et **ConvFloat** pour la conversion des entiers en flottants.

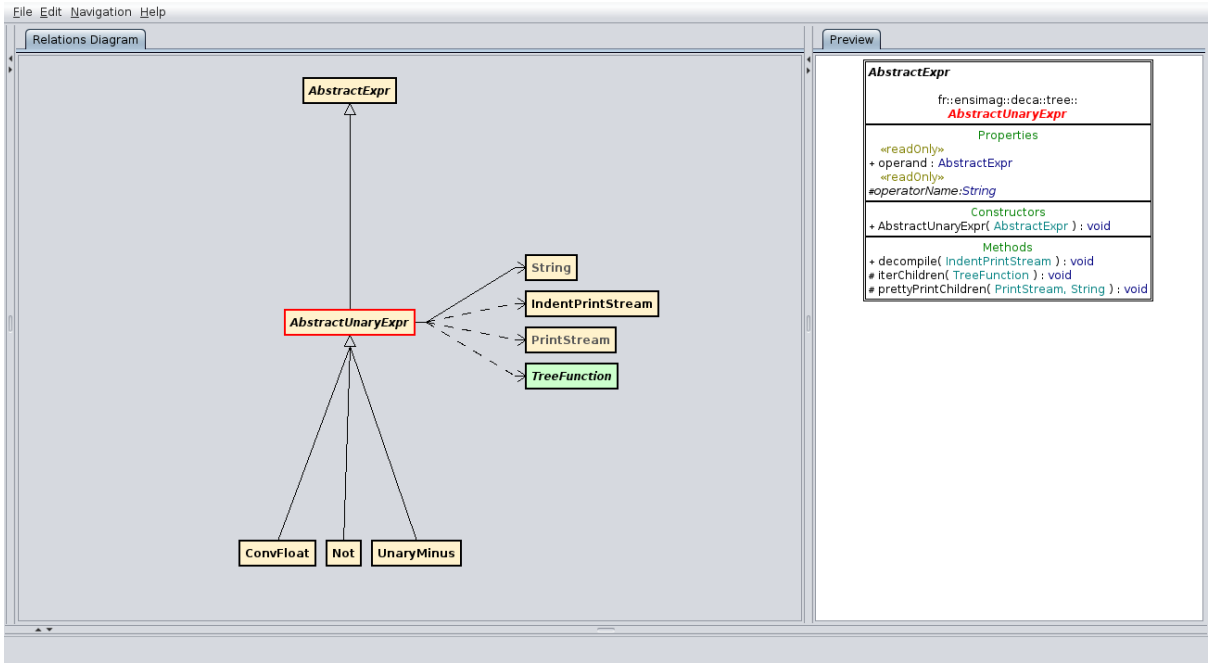

FIGURE 21 – UML diagram of **AbstractUnaryExpr**

1.3.13 AbstractLValue

Le terme *LValue* ou *left value* correspond au membre de gauche dans une expression. Ainsi la classe **AbstractLValue** est implémentée par **Selection** et **AbstractIdentifier**.

1.3.14 AbstractIdentifier

Cette classe est centrale dans l'implémentation du compilateur car elle traite les identifiants de type et d'expressions au moyen des méthodes *verifyType* et *verifyExpr*. De nombreuses classes possèdent des attributs cette classe et font ainsi appel à ces méthodes dans leurs propres vérifications afin de synthétiser le type de leurs identifiants.

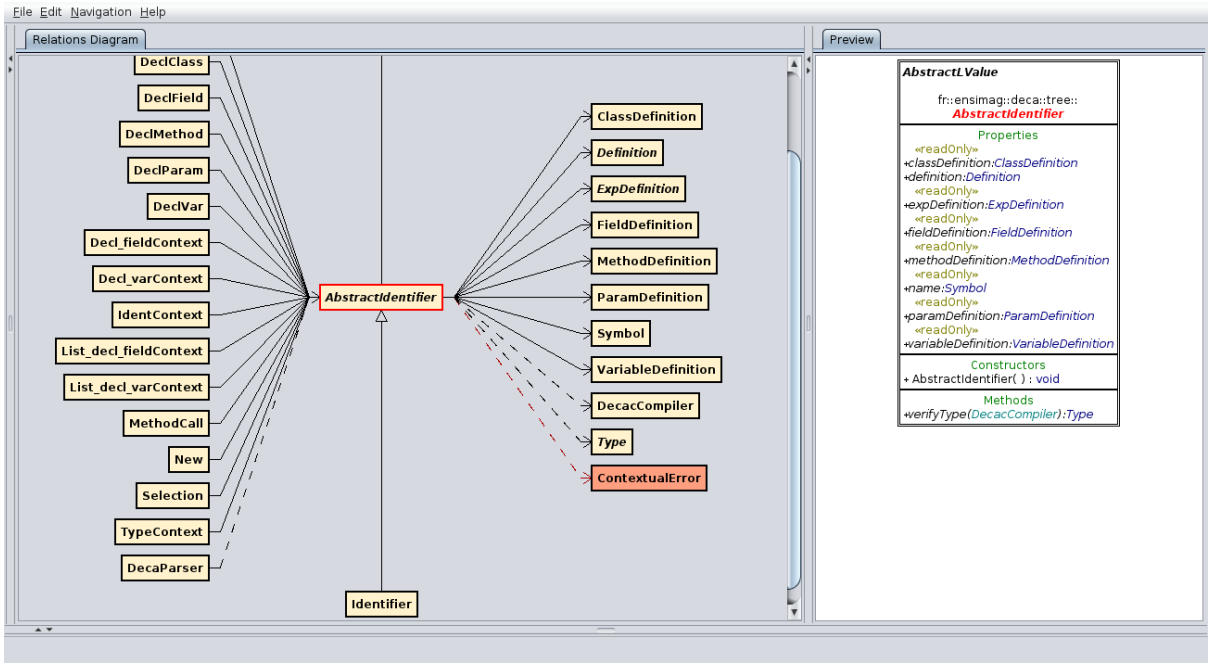

FIGURE 22 – UML diagram of **AbstractIdentifier** – used by (left) / uses (right)

1.3.15 Selection

Cette classe implémente le point . permettant la sélection d'un champ *fieldName* à partir d'un objet *objectName*. On vérifie alors que ce champ existe dans la classe de l'objet et que sa visibilité y autorise l'accès.

1.3.16 AbstractReadExpr

Cette classe abstraite est implémentée par deux sous classes **ReadInt** et **ReadFloat** afin de lire les entiers et flottants que l'utilisateur saisi.

1.3.17 Literal

Les 4 terminaux **StringLiteral**, **IntLiteral**, **FloatLiteral** et **BooleanLiteral** permettent le typage des valeurs numériques, chaînes de caractères et booléens.

1.3.18 MethodCall

Un appel de méthode en Deca est traité par **MethodCall** afin de vérifier l'existence de *methodName* et que les expressions constituant les arguments de la méthode sont licites.

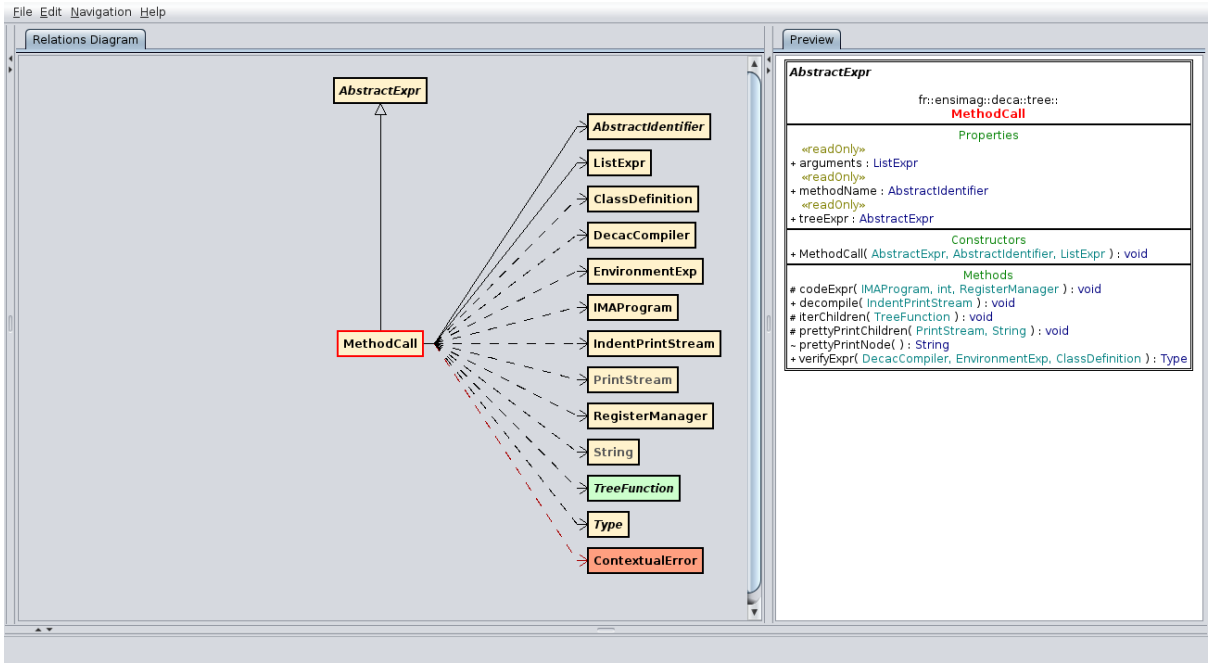

FIGURE 23 – UML diagram of **MethodCall**

1.3.19 New

Implémentation du terminal associé à *new Xyz()*, on vérifie que *Xyz* est une classe définit dans l'environnement des types.

1.3.20 This

Implémentation du terminal associé à *this*, afin d'utiliser un champ ou méthode dans une classe.

1.3.21 Null

Implémentation du terminal associé à *null*, on le décore du type nul.

1.3.22 DeclVar

Implémentation du non terminal associé aux déclarations *type varName initialization ;* dans la fonction *Main* du programme Deca. On vérifie si la déclaration est conforme aux règles de la syntaxe contextuelle du langage.

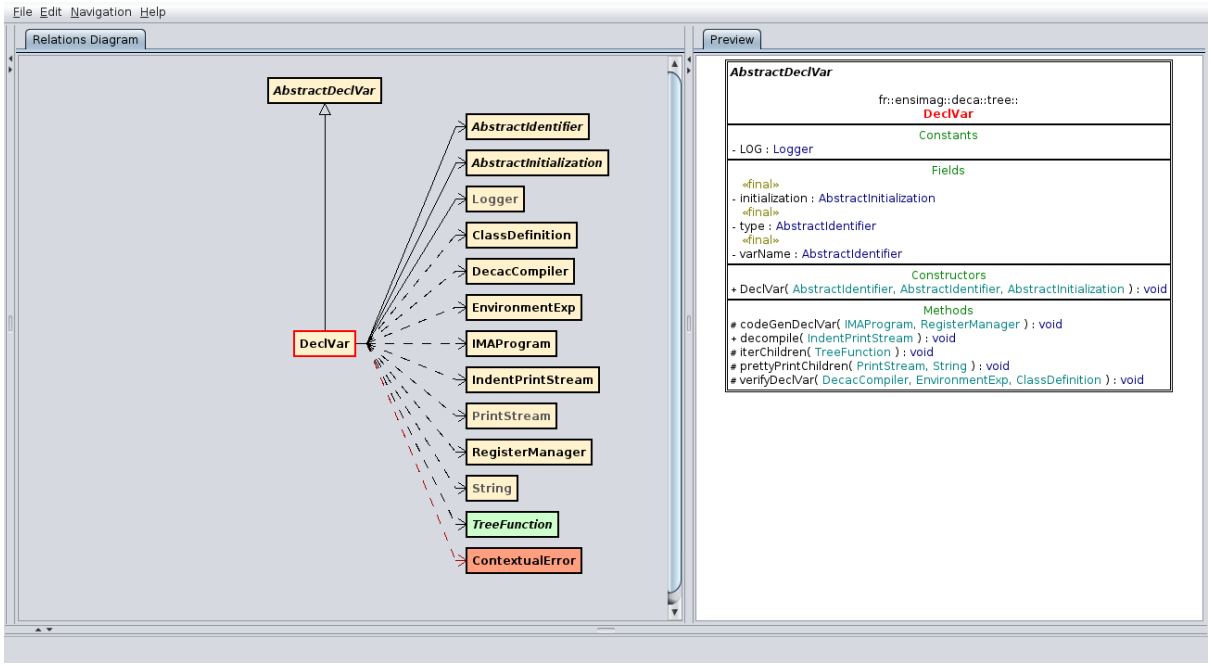

FIGURE 24 – UML diagram of **DeclVar**

1.3.23 DeclClass

Implémentation du non terminal associé aux déclarations *class className extends superClassName {}*. On réalise trois passes de vérification à partir de ce noeuds :

1. *verifyClass* vérifie que la déclaration est licite.
2. *verifyClassMembers* vérifie que les champs et méthodes sont licites.
3. *verifyClassBody* vérifie que les instructions et expressions sont licites.

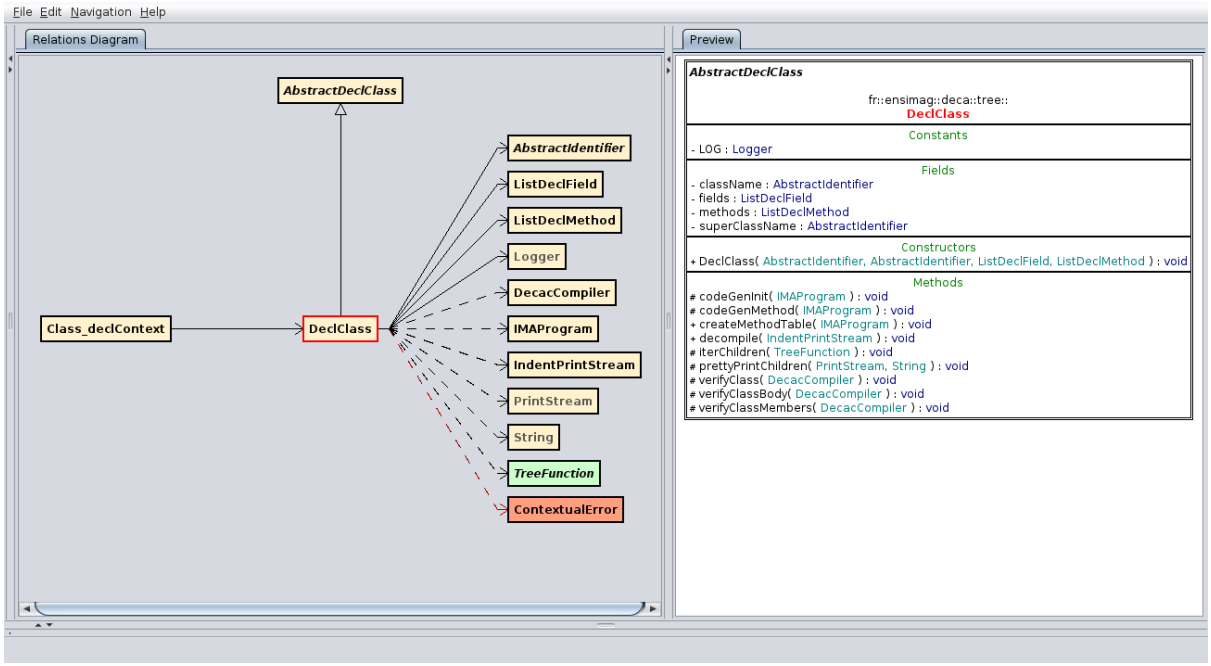

FIGURE 25 – UML diagram of **DeclClass**

1.3.24 DeclField

Implémentation du non terminal associé aux déclarations de champs d'une classe. Les méthodes *verifyField* et *verifyClassBodyField* vérifient respectivement les identifiants et l'initialisation.

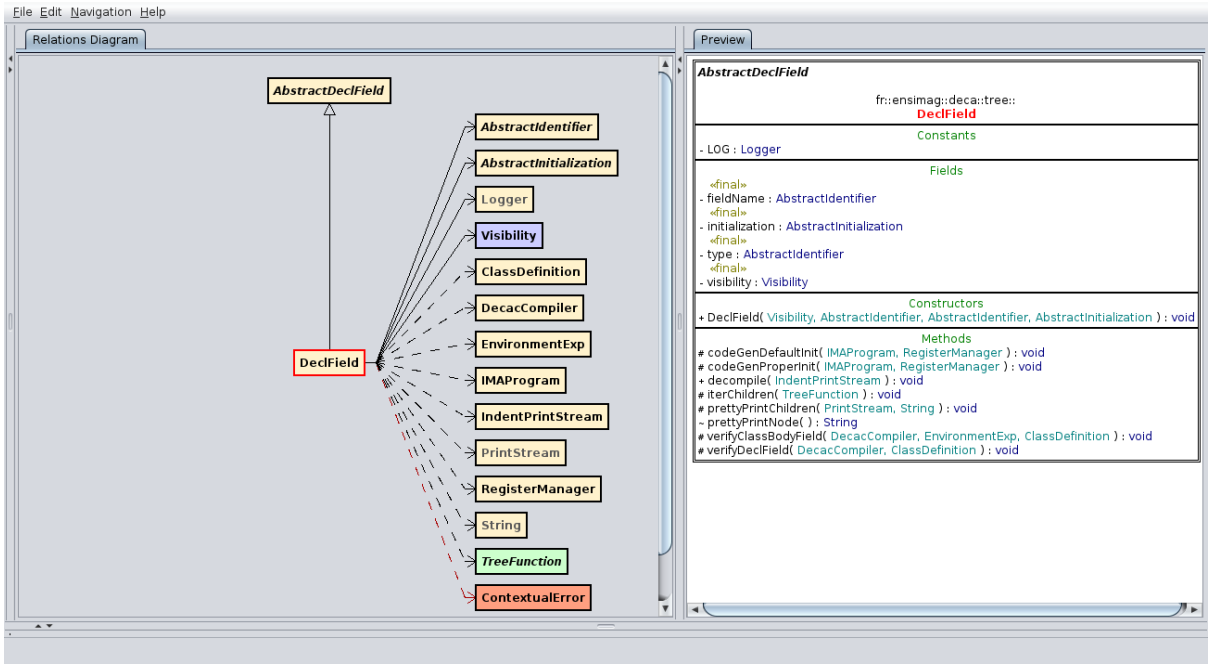

FIGURE 26 – UML diagram of **DeclField**

1.3.25 DeclMethod

Implémentation du non terminal associé aux déclarations de méthodes d'une classe. Les méthodes *verifyDeclMethod* et *verifyClassBodyMethod* vérifient respectivement les identifiants (et l'éventuelle redéfinition) et le corps de la méthode.

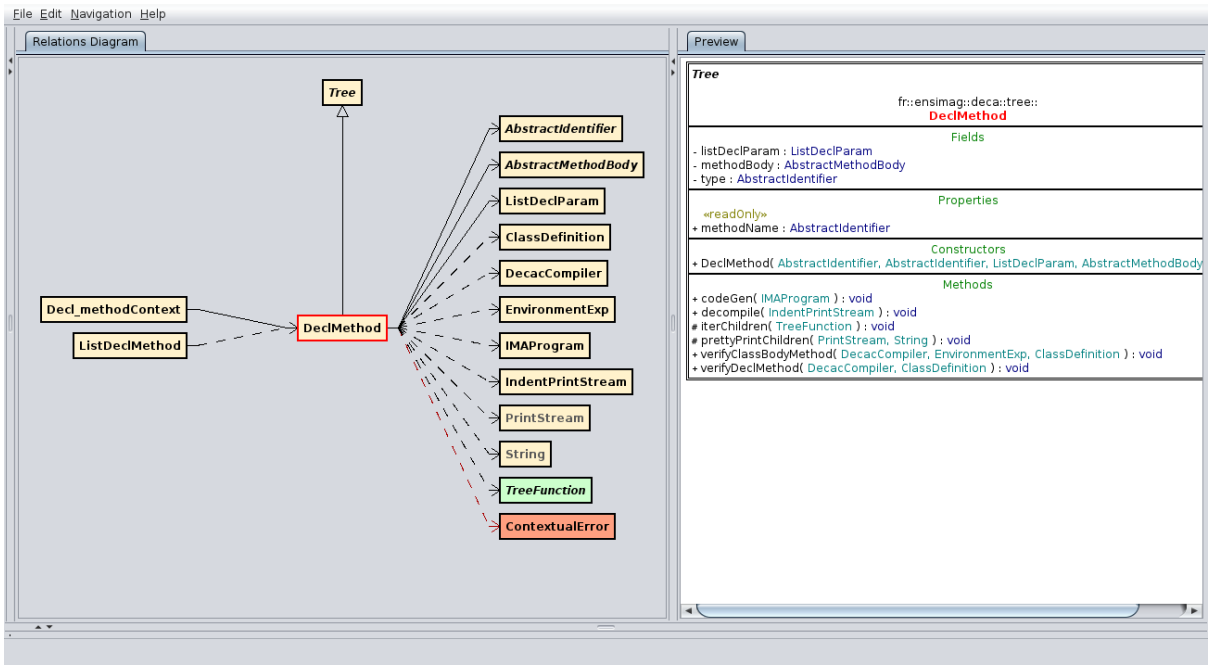

FIGURE 27 – UML diagram of **DeclMethod**

1.3.26 DeclParam

Implémentation du non terminal associé aux déclarations des paramètres d'une méthode. Les méthodes *verifyDeclParam* et *verifyClassBodyParam* vérifient respectivement les identifiants des types et les identifiants des noms des paramètres.

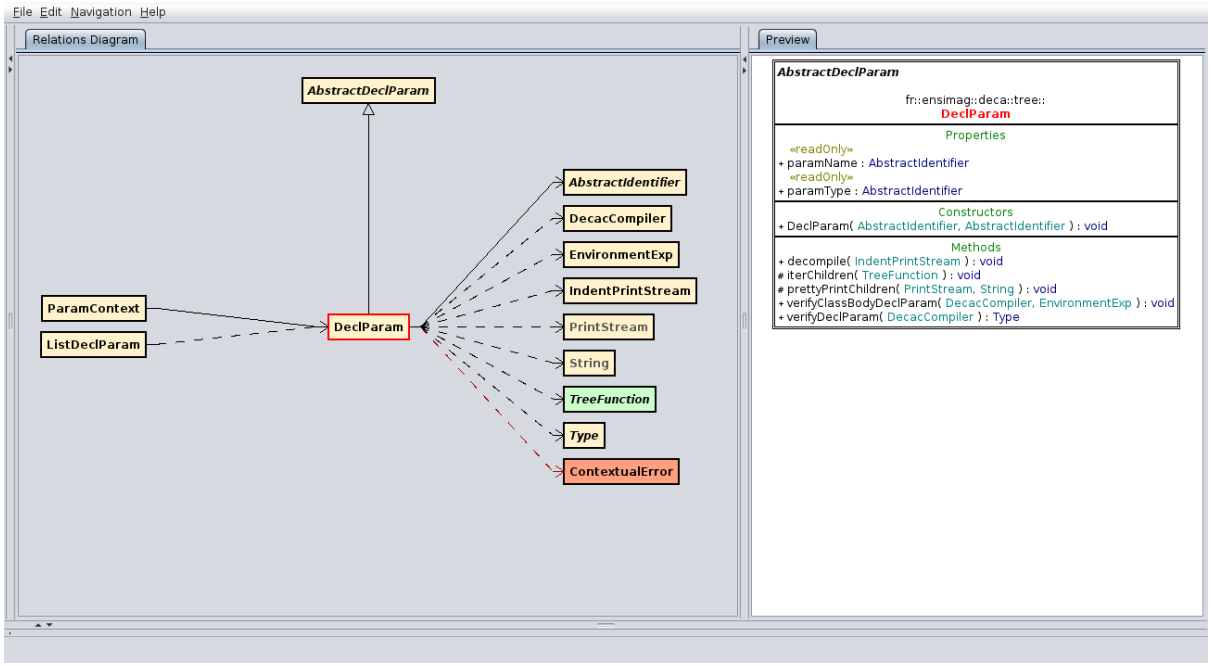

FIGURE 28 – UML diagram of **DeclParam**

1.3.27 AbstractInitialization

Classe correspondant aux initilialisations des variables et champs, on vérifie la cohérence de l'expression en fonction du type attendu.

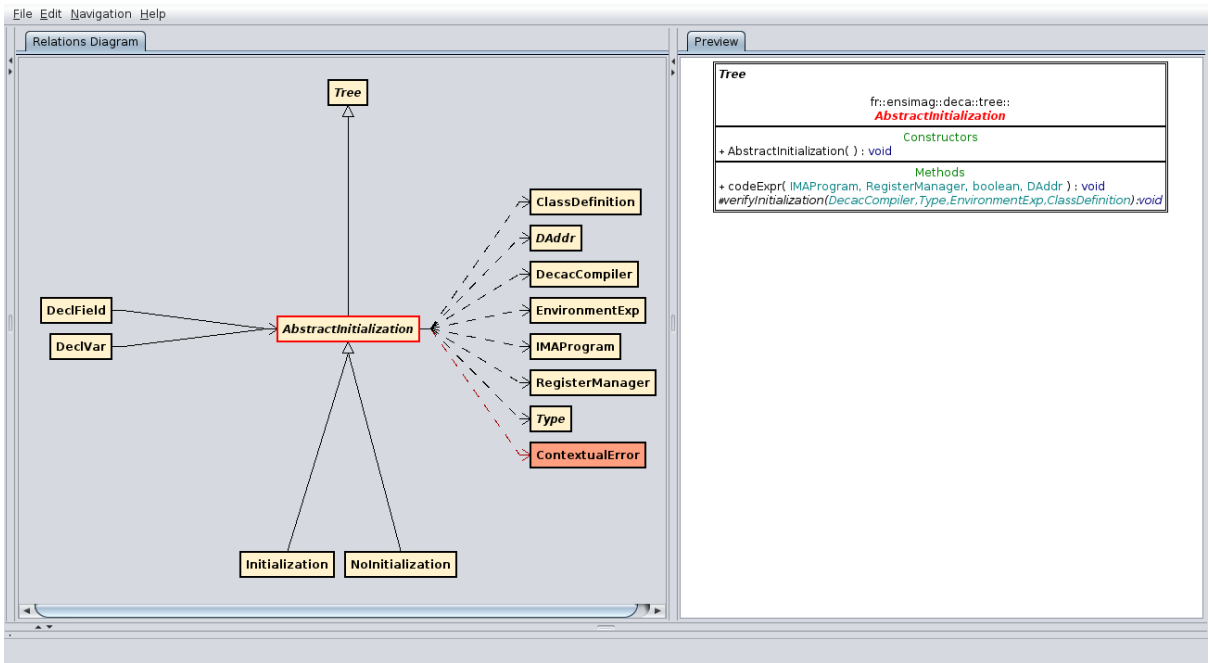

FIGURE 29 – UML diagram of **AbstractInitialization**

1.3.28 AbstractProgram

Point d'entrée de l'arbre associé à un programme Deca, on effectue les trois passes de la vérification contextuelle dans la méthode *verifyProgram*

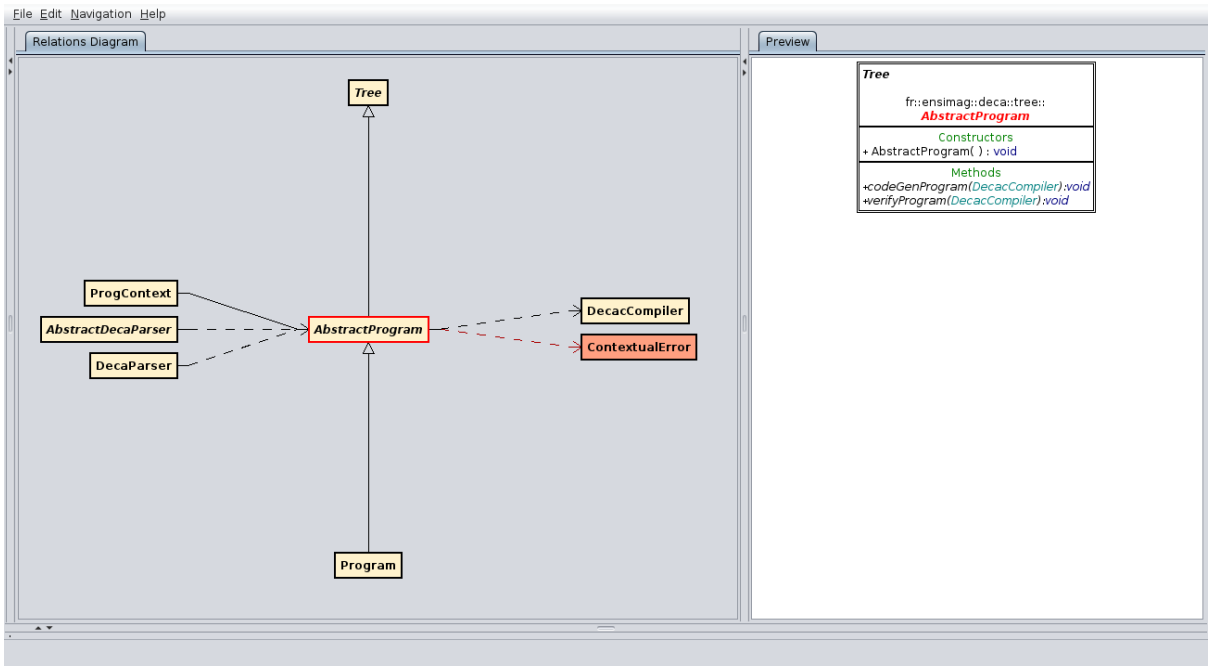

FIGURE 30 – UML diagram of **AbstractProgram**

1.3.29 AbstractMain

Correspond au *Main* principal du programme Deca, il contient les listes des déclarations de variables et des instructions.

1.3.30 AbstractMethodBody

Correspond au corps des méthodes des classes du programme Deca, il contient les listes des déclarations de variables et des instructions - ou un code assembleur.

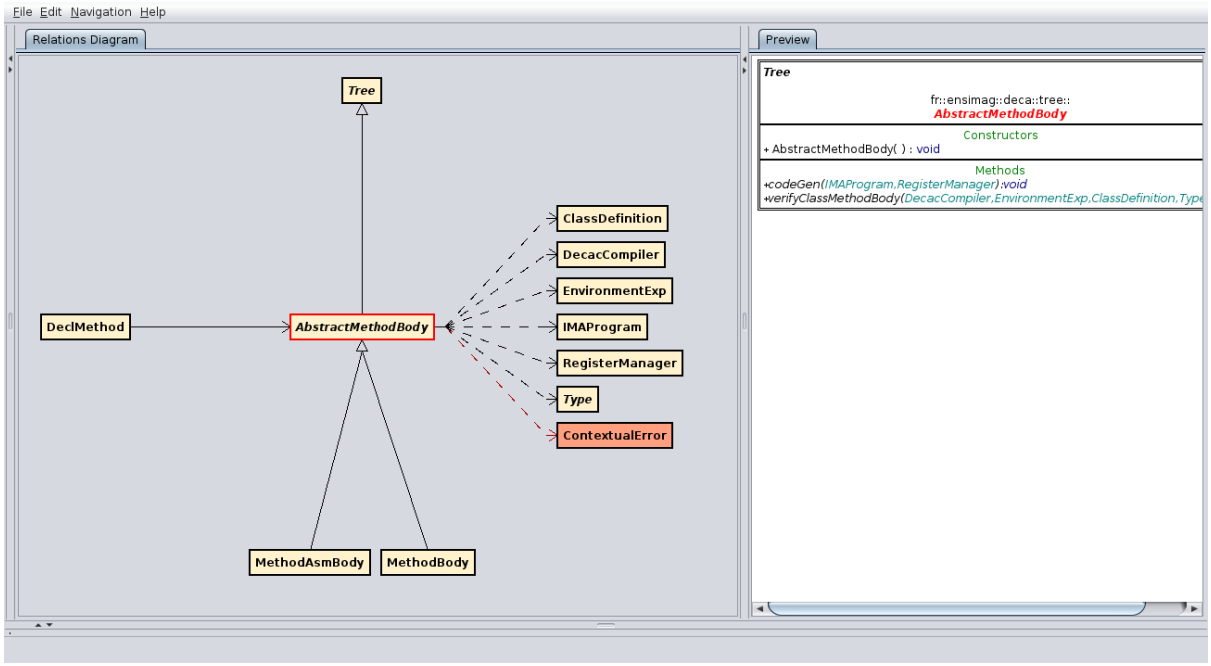

FIGURE 31 – UML diagram of **AbstractMethodBody**

1.4 Classes spécifiques à la génération de code assembleur

1.4.1 MethodTable

Ensemble de méthodes gérant la génération de la tables des méthodes.

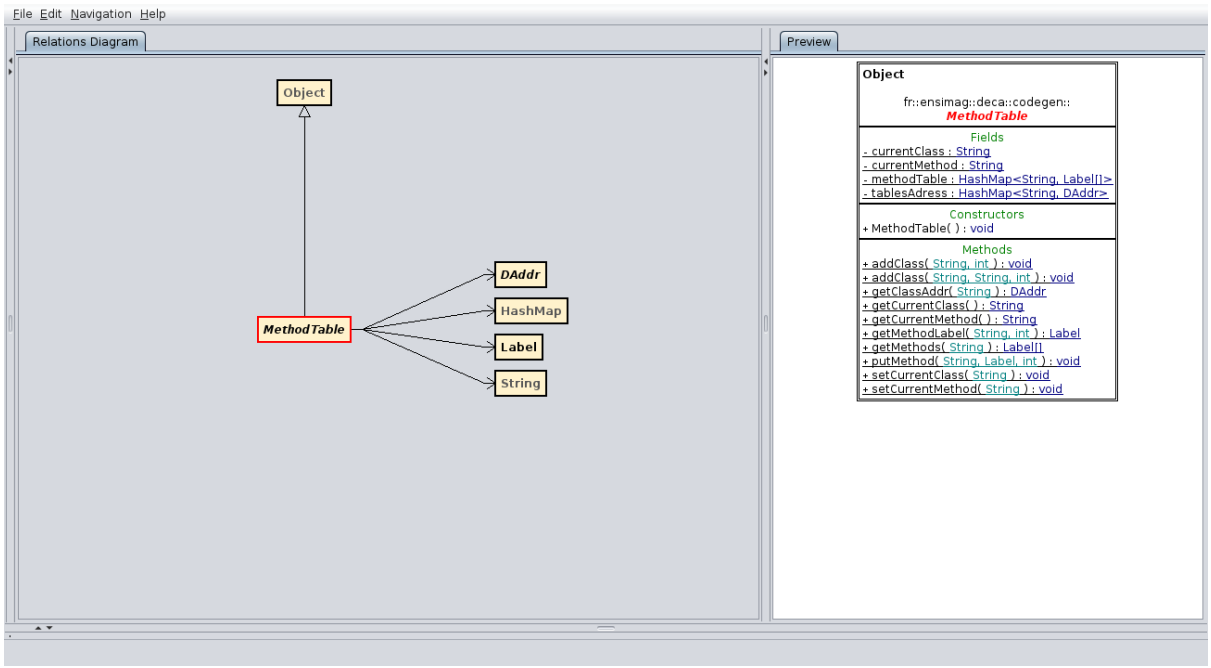

FIGURE 32 – UML diagram of **MethodTable**

1.4.2 RegisterManager

Ensemble de méthodes gérant l'adressage des variables locales, la sauvegarde des registres et la génération des instructions assembleur *TSTO* et *ADDSP* qui nécessitent de connaître le nombre de variables locales, et d'empilements effectués dans le bloc.

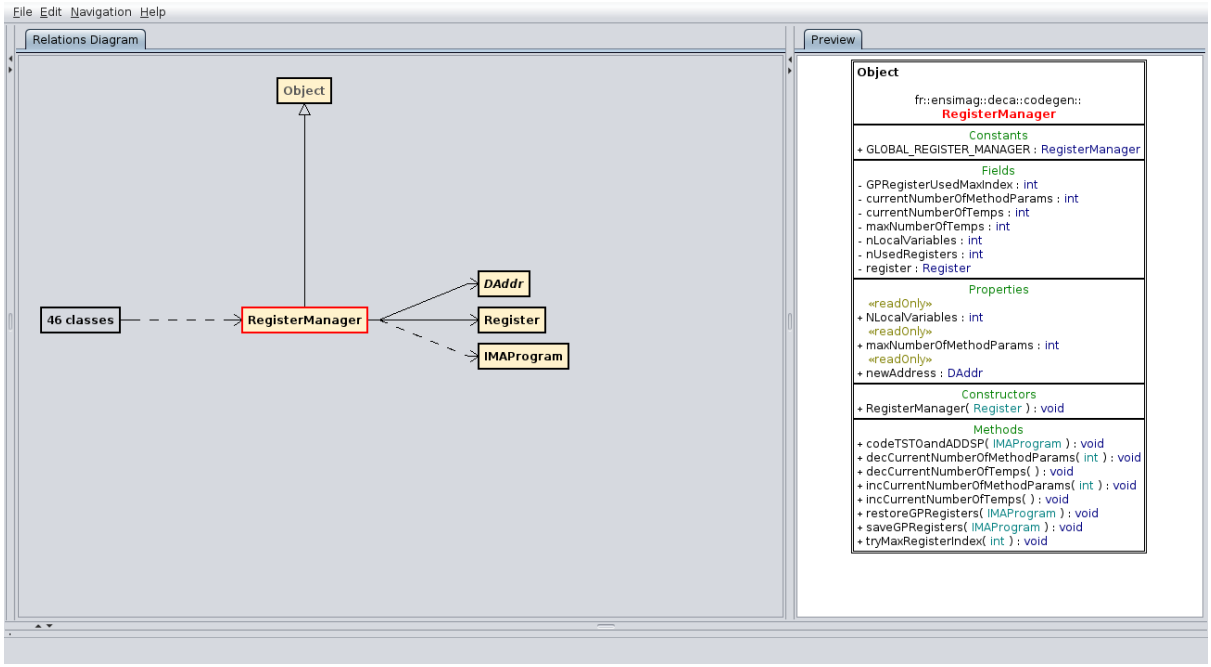

FIGURE 33 – UML diagram of **RegisterManager**

2 Spécifications sur le code du compilateur

2.0.1 Environnement des types

La table de symbole du parseur générant le symbole des types rencontrés et celle associée à **DecacCompiler** étant différente, on récupère les définitions des types prédéfinis à l'aide de la méthode *getDefinitionFromName* avec comme argument une chaîne de caractère car les symboles ne seraient pas égaux.

2.0.2 Assign compatible

Lors d'une affectation, de l'appel ou de la redéfinition d'une méthode, on procède à une vérification des types selon la règle suivante :

- si on attend un **FloatType** et que l'on a un **IntType**, on ajoute un noeud **ConvFloat** de type flottant en amont de l'expression entière.
- si le type attendu est **ClassType**, alors on vérifie que l'on a un **Nulltype** ou un **ClassType** tel que ce type correspond à une sous classe. La méthode *isSubClassOf* remonte ainsi la hiérarchie des classes à partir du champ super-classe des **ClassDefinition** successives.

2.0.3 Cast & instanceOf

Cette version du compilateur ne gère pas le cast explicite et les méthodes instanceOf ne sont pas implémentées.

2.0.4 Bugs MethodCall

Il faut noter que le comportement d'un appel de méthode dans le cas ou le nombre d'arguments n'est pas celui attendu. Il compile et produit des erreurs si ce nombre est inférieur à celui attendu. Si il est supérieur, les arguments supplémentaires sont négligés. En outre, les conversions implicites de type *assign compatible* sont acceptées mais pas effectuées.