Document Architecture Logicielle
Projet UML Reverse

29 janvier 2016

Version :	0.1
Date :	29 janvier 2016
Rédigé par :	Yohann Henry Nabil Belkhous Stephen Cauchois Anthony Godin Florian Inchingolo Saad Mrabet Nicolas Meniel

Mises à jour

Version	**Date**	**Modifications réalisées**
0.2	22/01/2016	Architecture refaite dans l'intégralité
0.1	14/01/2016	Première version

Table des matières

1 Objectif

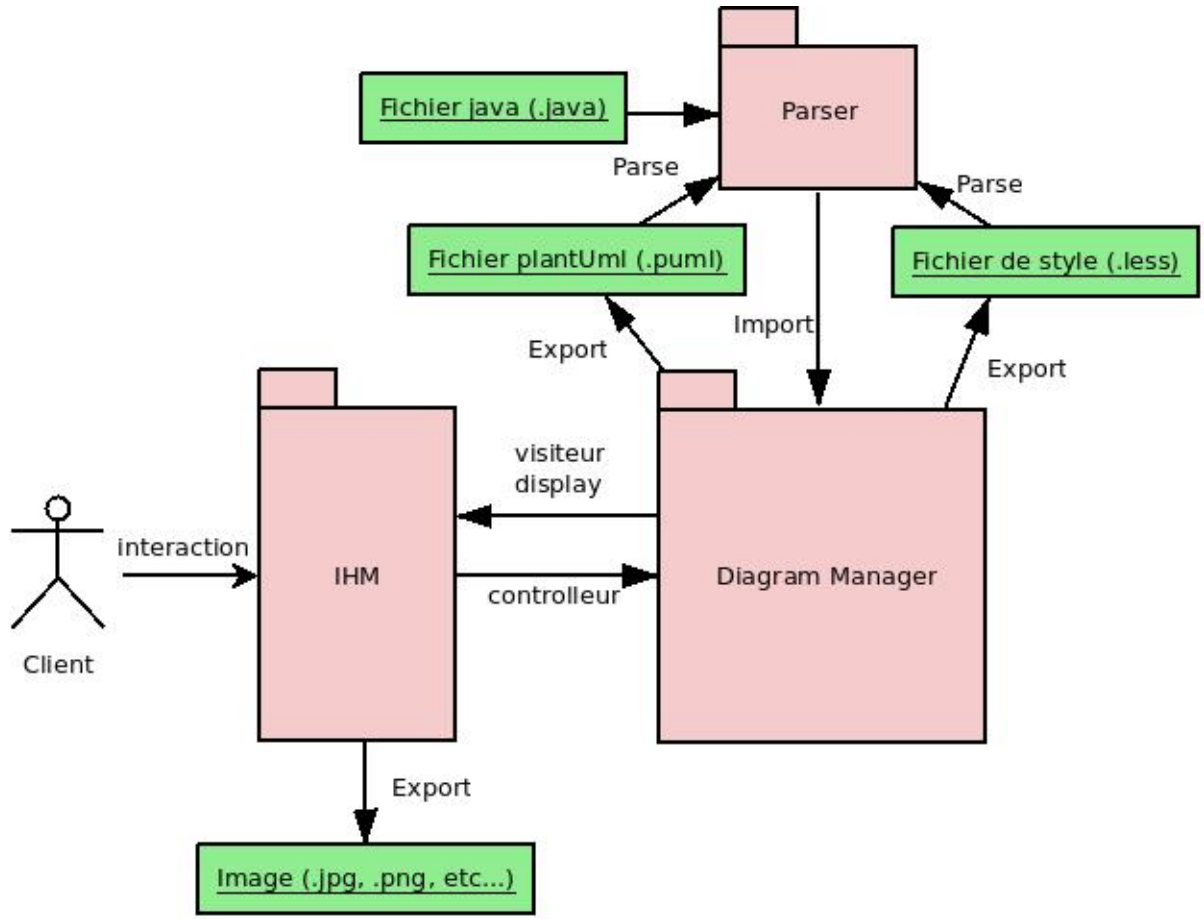
Ce document représente la structure générale du logiciel et les modèles de conception mis en oeuvre pour le réaliser. Il est destiné aux membres de l'équipe de développment, notamment aux concepteurs, ainsi qu'aux superviseurs du projet.

2 Les technologies utilisées

Nous allons utiliser différentes technologies pour la construction du projet.
Le projet est développé en java 1.8. Le projet utilise :

- *Dot* : Un outil permettant de calculer la position idéale des entités et des relations d'un graphe. Cela nous évite la partie mathématique pour le placement des relations qui aurait pu s'avérer un risque majeur.
- *JUnit* : Framework pour valider chaque classe par le biais de tests unitaires.
- *Maven* : Un outil pour la gestion des dépendances de l'application.
- *Antlr* : Un parser dans lequel nous pourrons définir les grammaires pour l'extraction des données d'un fichier java ou plantUml.
- *openJFX* : Une bibliothèque libre graphique parfaite pour ce projet. La bibliothèque permet d'associer à une entité du CSS, ce qui simplifie fortement notre travail.
- *SceneBuilder* : Logiciel permettant de créer des vus de façon ergonomique en drag and drop. Les vus sont créées en fxml.

3 Fonctionnement général



L'architecture est construite suivant le modèle MVC. Cela nous permet de séparer les responsabilités des classes.

4 Organisation des paquetages

- src
  - main — Contient la classe d'entrée du logiciel
    - UmlReverseApp.java
  - model
    - diagram — Modèlise un diagramme en objet
      - class
        - relation
        - visitor — Permet le parcourt du diagramme de classe
      - usecase
        - relation
        - visitor — Permet le parcourt du diagramme de cas
      - util — Contient les classes communes aux diagrammes
      - visitor — Visiteurs globaux. Choisi d'appeller un visiteur en particulier selon le type de diagramme
    - io
      - antlr
    - filemanagerr — Contient les classes gérant la gestion des dossiers de projet
  - ui
    - view — Contient les vues principales de l'application avec leurs contrôleurs
    - component — Contient les composants utilisés dans la vue
      - class
      - usecase
      - common
      - relation
        - class
        - usecase
        - common

5 Paquetage model

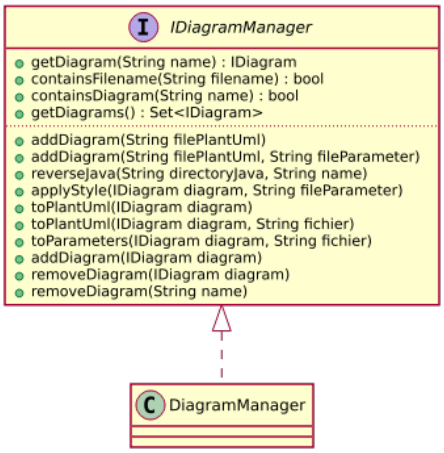
Ce package contient l'intégralité du modèle.

C'est à dire le gestionnaire de diagramme (*IDiagramManager*) ainsi que le gestionnaire de fichier (*TreeFileManager*).

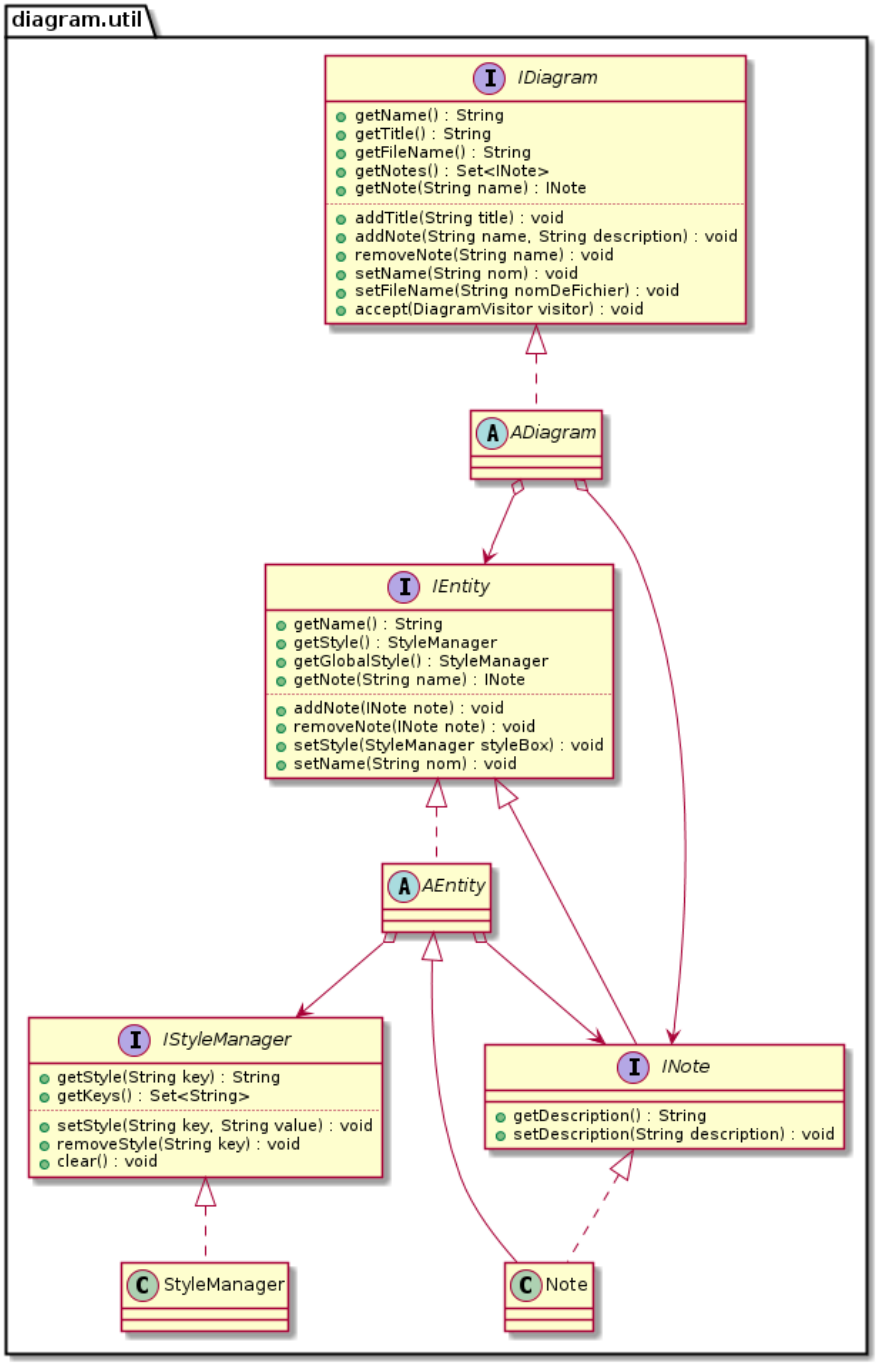
5.1 diagram

IDiagram est l'interface qui représente tous les types de diagrammes existants.

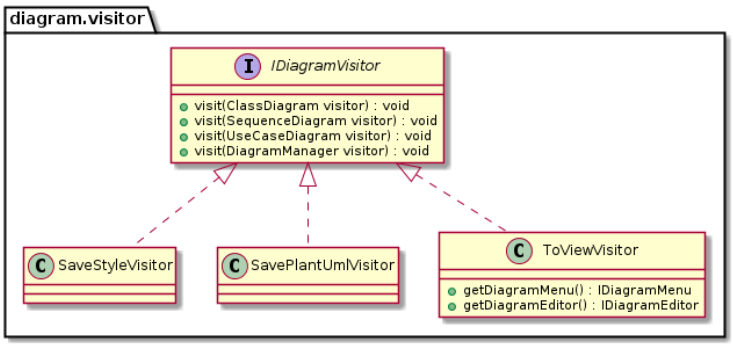
Manager Le *IDiagramManager* est la façade qui stocke et gère les différents diagrammes. Il est aussi la façade pour le *parser*. Il s'occupe de sérialiser à l'aide des visiteurs les différents diagrammes.


IDiagramManager
getDiagram(String name) : IDiagram
containsFilename(String filename) : bool
containsDiagram(String name) : bool
getDiagrams() : Set<IDiagram>
addDiagram(String filePlantUml)
addDiagram(String filePlantUml, String fileParameter)
reverseJava(String directoryJava, String name)
applyStyle(IDiagram diagram, String fileParameter)
toPlantUml(IDiagram diagram)
toPlantUml(IDiagram diagram, String fichier)
toParameters(IDiagram diagram, String fichier)
addDiagram(IDiagram diagram)
removeDiagram(IDiagram diagram)
removeDiagram(String name)
DiagramManager

util Ce package contient les différentes classes utilisées par tous les diagrammes. On y retrouve le maximum de code commun comme les *INotes*, les gestionnaires de style et quelques classes abstraites qui peuvent être héritées pour récupérer la mécanique de base des entitées ou des diagrammes.

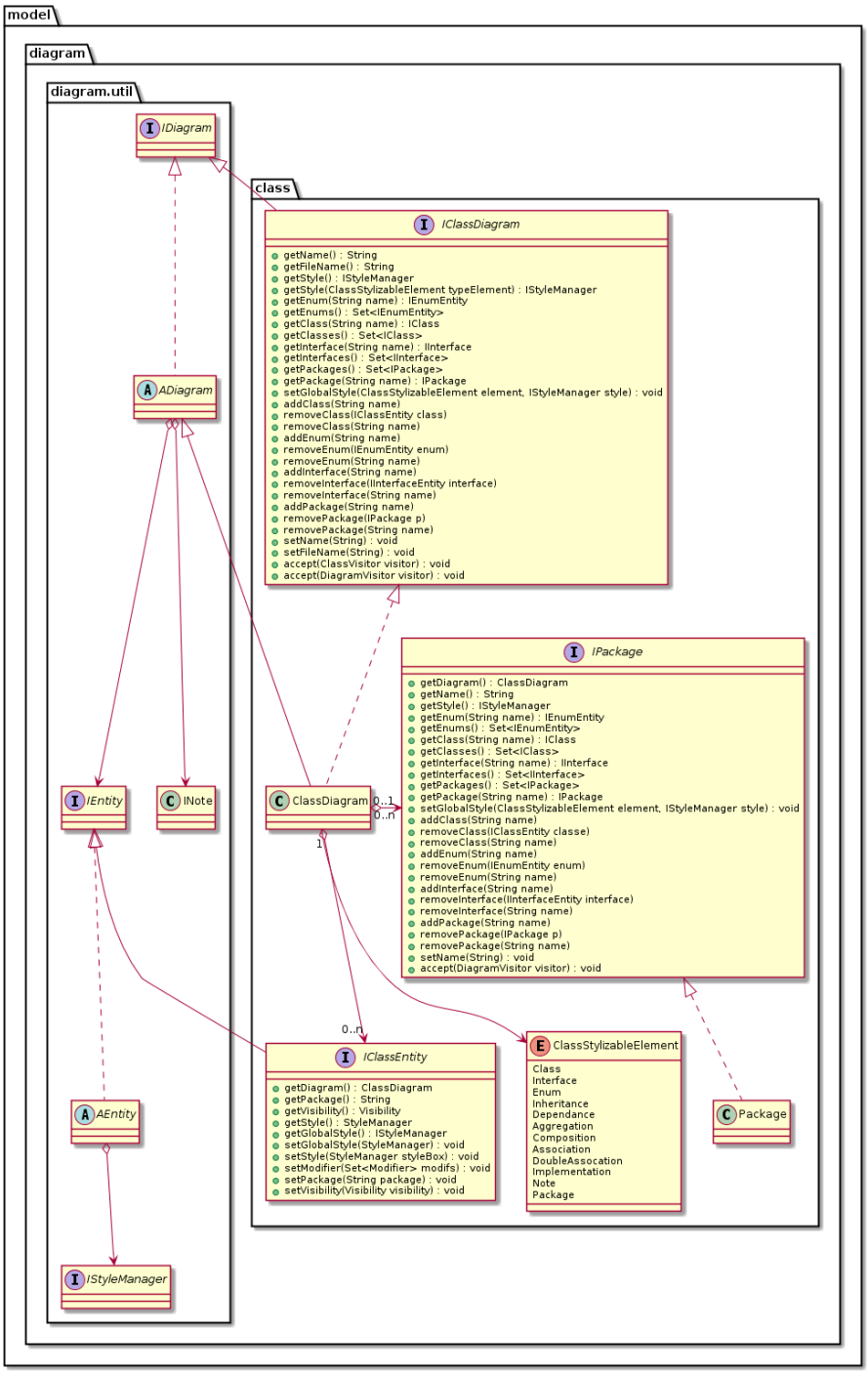


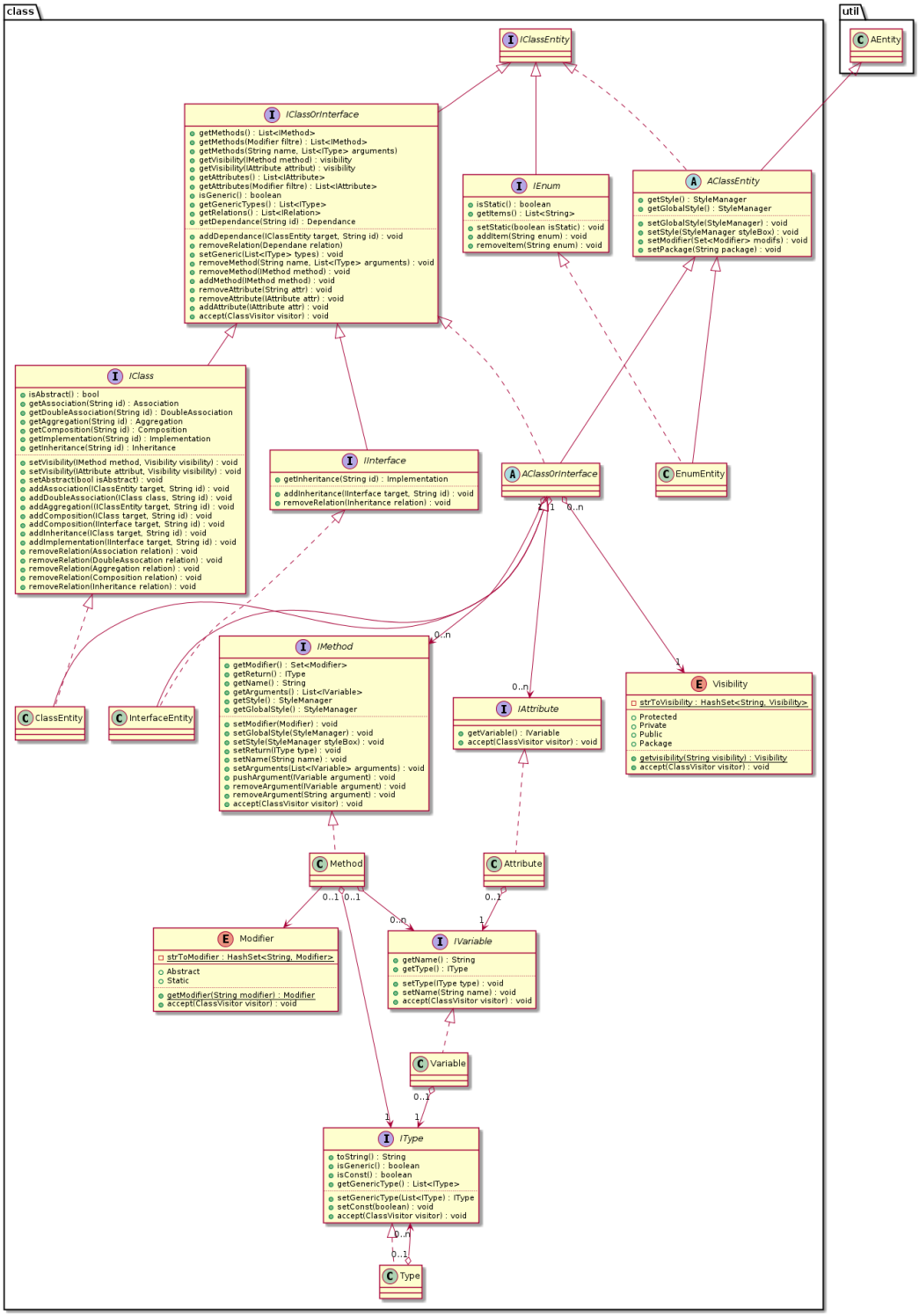
Visiteur Il s'agit, dans ce package, de visiteurs qui permettent de lancer sur un diagramme les visiteurs spécialisés pour le typage du diagramme. Cela nous permet de lancer les visiteurs sans connaitre le typage d'un diagramme. Les visiteurs permettent de sérialiser un diagramme en fichier de paramètre, de style ou de créer la vue correspondante. Chaque diagramme possède son propre package *visiteur* avec les visiteurs propres à exécuter ces fonctionnalités pour le type correspondant.



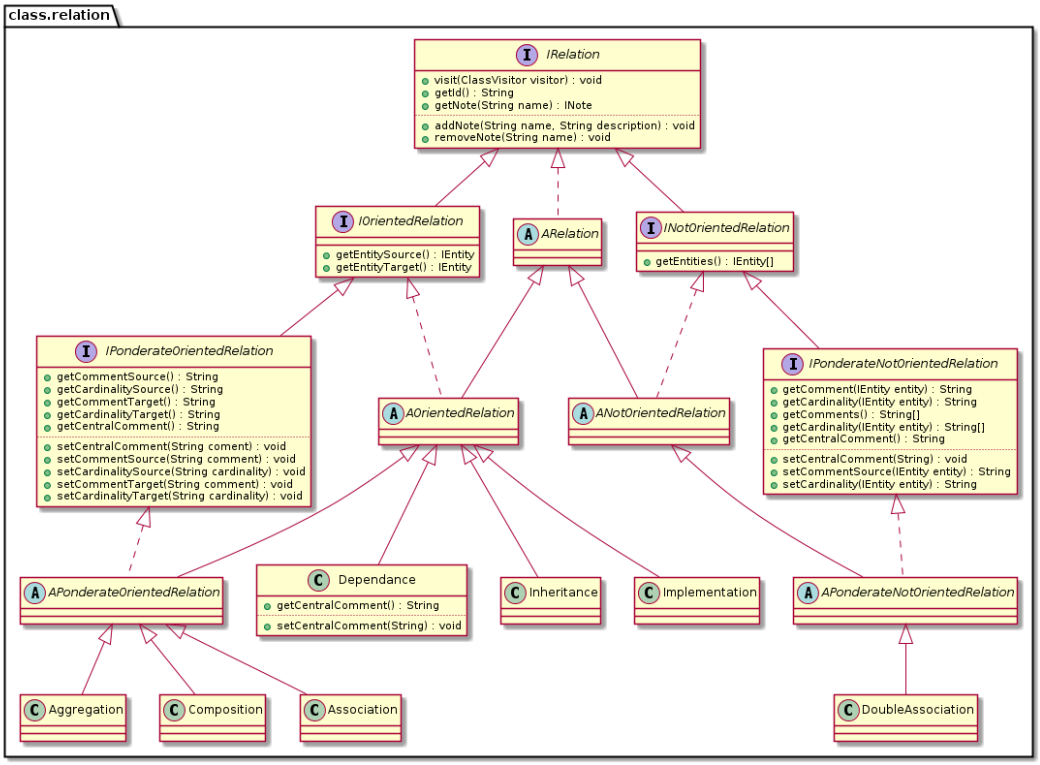
IDiagramClass Cette classe modélise un diagramme de classe en objet. Il est possible d'effectuer tout un tas d'opérations dessus qui vérifient les exigences MOD_10 MOD_20 MOD_30 MOD_40 MOD_50 MOD_60 MOD_70 MOD_80 MOD_90 MOD_100 MOD_110 de la STB.

Les entités Les différentes entitées d'un diagramme de classe sont les énumérations(*I*Enum), les classes(*IClass*) et les interfaces(*IInterface*). Elles possèdent chacune beaucoup de code commun tout en ayant de grandes disparités de comportements. Les énumérations possèdent une liste d'élément. Les classes et les interfaces possèdent des méthodes(*IMethod*) ainsi que des attributs(*IAttribute*). Les classes associent à chaque méthode et attribut une visibilité (*IVisibility*). Les fonctions sont composées d'un nom, d'un type de retour(*IType*) ainsi que d'une liste de variable(*IVariable*). Les attributs contient une variable. Les variables contiennent un type et un nom. Un type possède un nom et, s'il est générique, une liste de type.

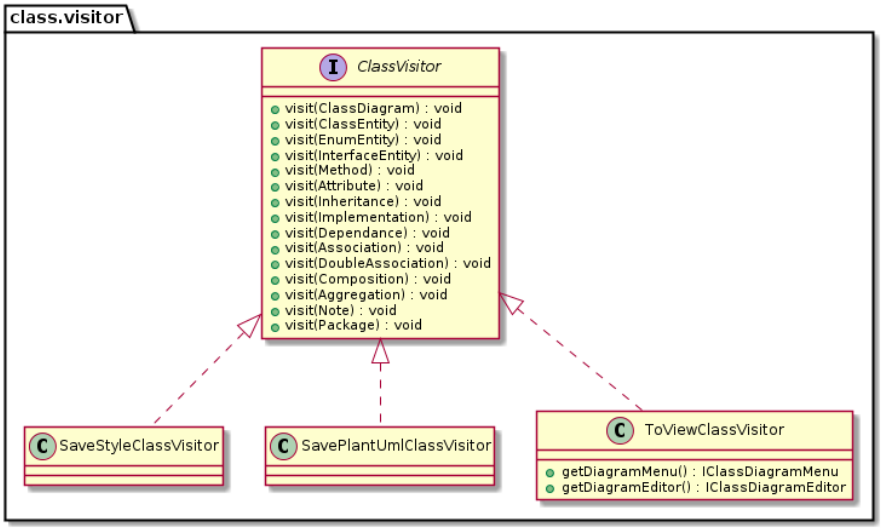



class
util
AEntity
IClassEntity
IClass0rInterface
getMethods() : List<IMethod>
getMethods(Modifier filtre) : List<IMethod>
getMethods(String name, List<IType> arguments)
getVisibility(IMethod method) : visibility
getVisibility(IAttribute attribut) : visibility
getAttributes() : List<IAttribute>
getAttributes(Modifier filtre) : List<IAttribute>
isGeneric() : boolean
getGenericTypes() : List<IType>
getRelations() : List<IRelation>
getDependance(String id) : Dependance
addDependance(IClassEntity target, String id) : void
removeRelation(Dependane relation)
setGeneric(List<IType> types) : void
removeMethod(String name, List<IType> arguments) : void
removeMethod(IMethod method) : void
addMethod(IMethod method) : void
removeAttribute(String attr) : void
removeAttribute(IAttribute attr) : void
addAttribute(IAttribute attr) : void
accept(ClassVisitor visitor) : void
IEnum
isStatic() : boolean
getItems() : List<String>
setStatic(boolean isStatic) : void
addItem(String enum) : void
removeItem(String enum) : void
AClassEntity
getStyle() : StyleManager
getGlobalStyle() : StyleManager
setGlobalStyle(StyleManager) : void
setStyle(StyleManager styleBox) : void
setModifier(Set<Modifier> modifs) : void
setPackage(String package) : void
IClass
isAbstract() : bool
getAssociation(String id) : Association
getDoubleAssociation(String id) : DoubleAssociation
getAggregation(String id) : Aggregation
getComposition(String id) : Composition
getImplementation(String id) : Implementation
getInheritance(String id) : Inheritance
setVisibility(IMethod method, Visibility visibility) : void
setVisibility(IAttribute attribut, Visibility visibility) : void
setAbstract(bool isAbstract) : void
addAssociation(IClassEntity target, String id) : void
addDoubleAssociation(IClass class, String id) : void
addAggregation(IIClassEntity target, String id) : void
addComposition(IClass target, String id) : void
addComposition(IInterface target, String id) : void
addInheritance(IClass target, String id) : void
addImplementation(IInterface target, String id) : void
removeRelation(Association relation) : void
removeRelation(DoubleAssocation relation) : void
removeRelation(Aggregation relation) : void
removeRelation(Composition relation) : void
removeRelation(Inheritance relation) : void
IInterface
getInheritance(String id) : Implementation
addInheritance(IInterface target, String id) : void
removeRelation(Inheritance relation) : void
AClass0rInterface
EnumEntity
IMethod
getModifier() : Set<Modifier>
getReturn() : IType
getName() : String
getArguments() : List<IVariable>
getStyle() : StyleManager
getGlobalStyle() : StyleManager
setModifier(Modifier) : void
setGlobalStyle(StyleManager) : void
setStyle(StyleManager styleBox) : void
setReturn(IType type) : void
setName(String name) : void
setArguments(List<IVariable> arguments) : void
pushArgument(IVariable argument) : void
removeArgument(IVariable argument) : void
removeArgument(String argument) : void
accept(ClassVisitor visitor) : void
IAttribute
getVariable() : IVariable
accept(ClassVisitor visitor) : void
Visibility
strToVisibility : HashSet<String, Visibility>
Protected
Private
Public
Package
getvisibility(String visibility) : Visibility
accept(ClassVisitor visitor) : void
ClassEntity
InterfaceEntity
Method
Attribute
Modifier
strToModifier : HashSet<String, Modifier>
Abstract
Static
getModifier(String modifier) : Modifier
accept(ClassVisitor visitor) : void
IVariable
getName() : String
getType() : IType
setType(IType type) : void
setName(String name) : void
accept(ClassVisitor visitor) : void
Variable
IType
toString() : String
isGeneric() : boolean
isConst() : boolean
getGenericType() : List<IType>
setGenericType(List<IType) : IType
setConst(boolean) : void
accept(ClassVisitor visitor) : void
Type

Les relations Dans le package relation, on retrouve toutes les relations existantes pour le diagramme de classe.



Le visiteur

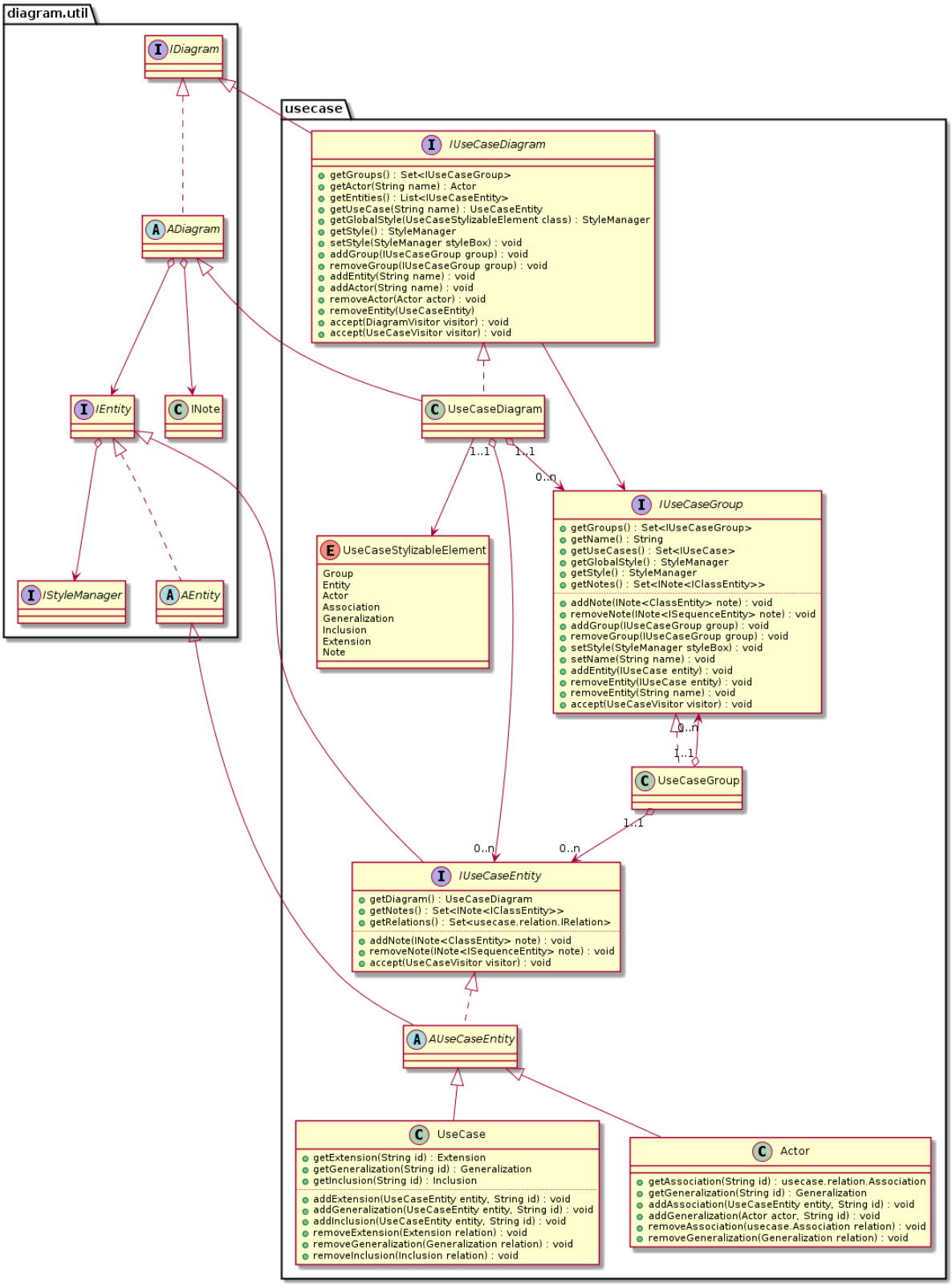


5.1.1 IDiagramUseCase

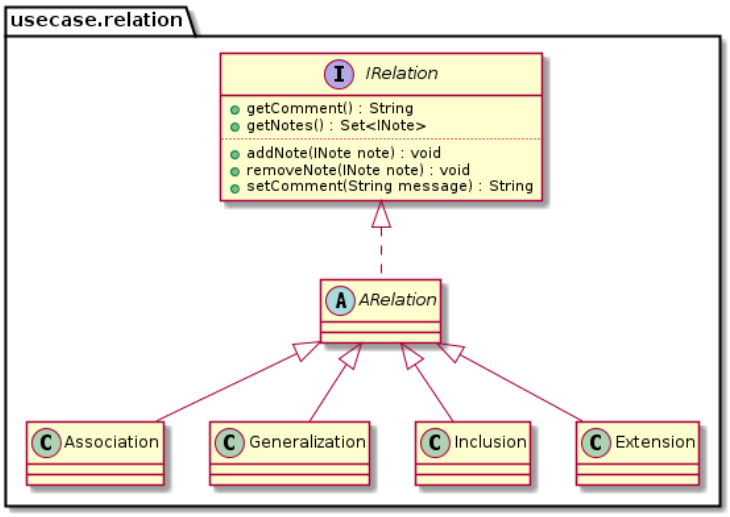
Cette classe modélise un diagramme de cas d'utilisation en objet. Il est possible d'effectuer tout un tas d'opérations dessus qui vérifient les exigences MOD_10 MOD_20 MOD_30 MOD_40 MOD_50 MOD_60 MOD_70 MOD_80 MOD_90 MOD_100 MOD_110 de la STB.

Les entités

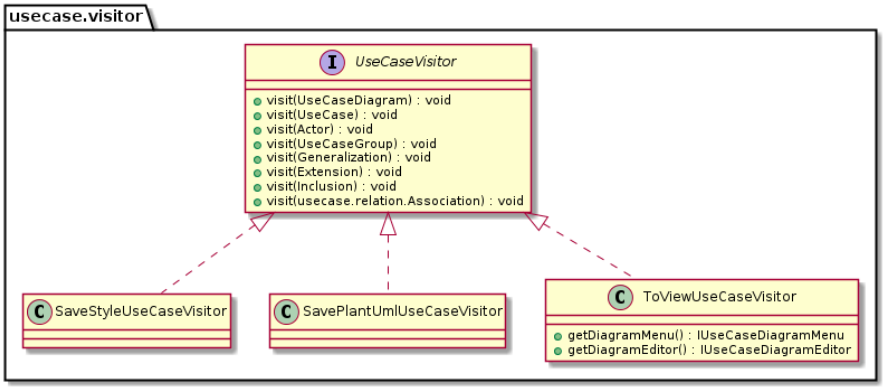
Le diagramme d'utilisation possède deux entités : Les acteurs(*Actor*) et les cas d'utilisation (*UseCase*). Chacun ne possède qu'une description et les relations partant d'elle vers une autre entité.



Les relations Dans le package relation, on retrouve les relations existantes pour le diagramme de cas d'utilisation.

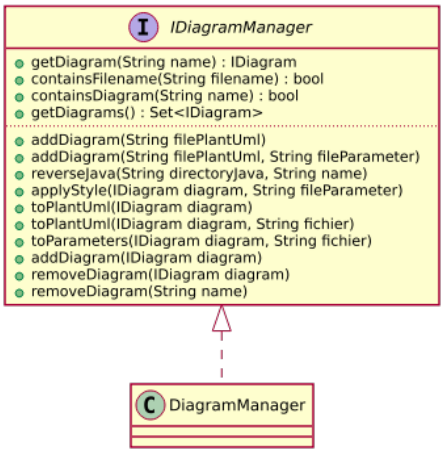


Le visiteur



5.2 IDiagramParser

5.2.1 Manager



5.2.2 util

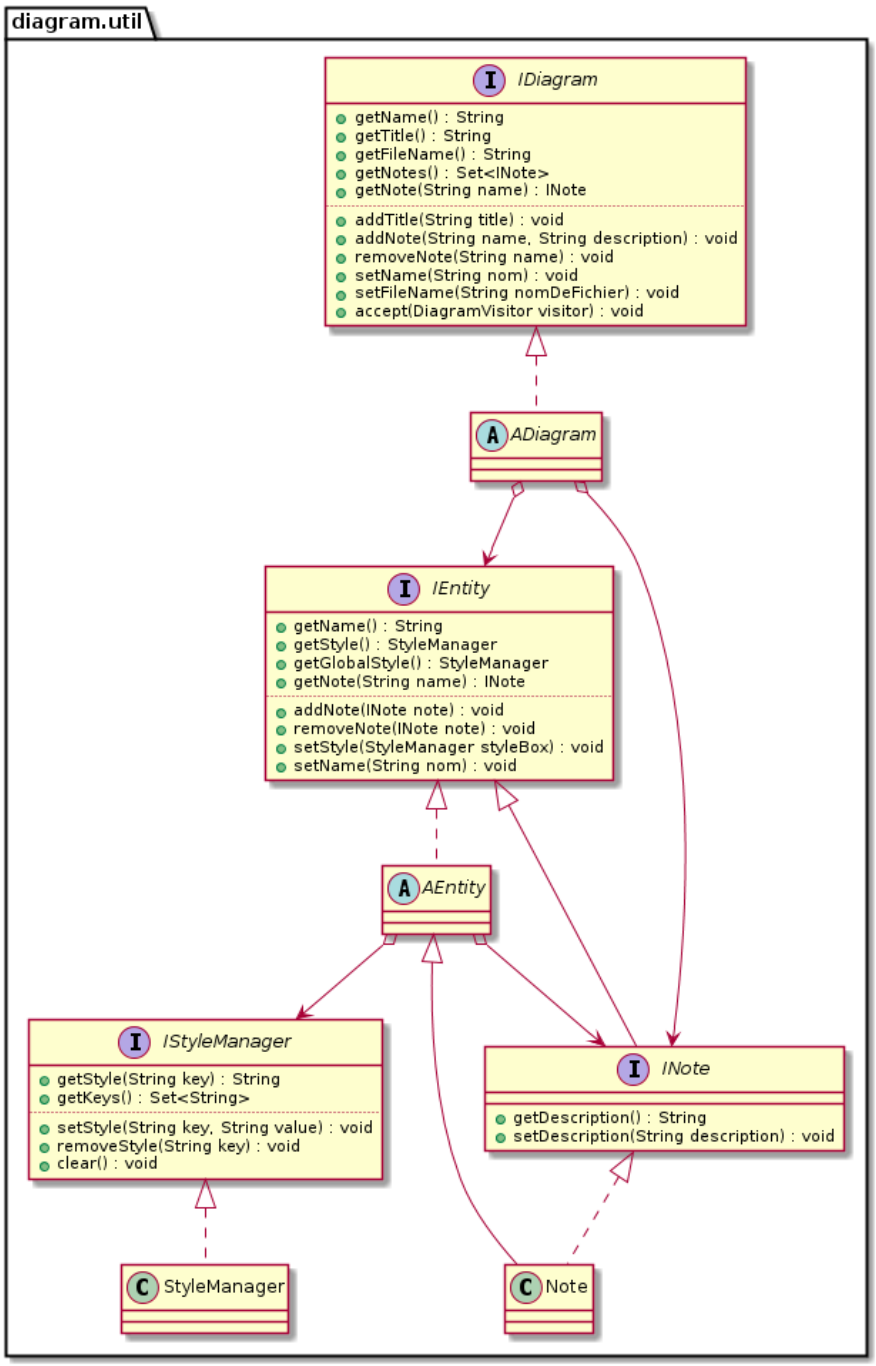
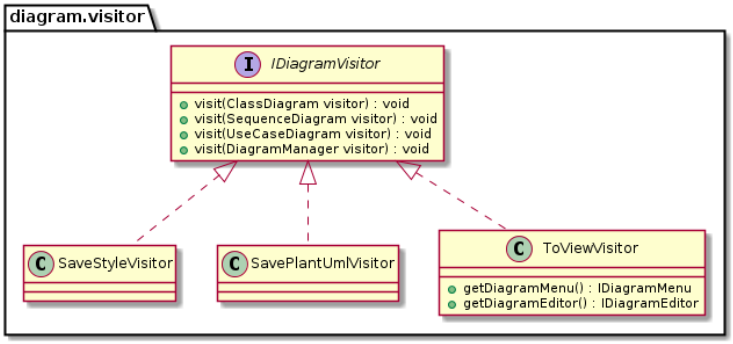


diagram.util
IDiagram
getName() : String
getTitle() : String
getFileName() : String
getNotes() : Set<INote>
getNote(String name) : INote
addTitle(String title) : void
addNote(String name, String description) : void
removeNote(String name) : void
setName(String nom) : void
setFileName(String nomDeFichier) : void
accept(DiagramVisitor visitor) : void
ADiagram
IEntity
getName() : String
getStyle() : StyleManager
getGlobalStyle() : StyleManager
getNote(String name) : INote
addNote(INote note) : void
removeNote(INote note) : void
setStyle(StyleManager styleBox) : void
setName(String nom) : void
AEntity
IStyleManager
getStyle(String key) : String
getKeys() : Set<String>
setStyle(String key, String value) : void
removeStyle(String key) : void
clear() : void
INote
getDescription() : String
setDescription(String description) : void
StyleManager
Note

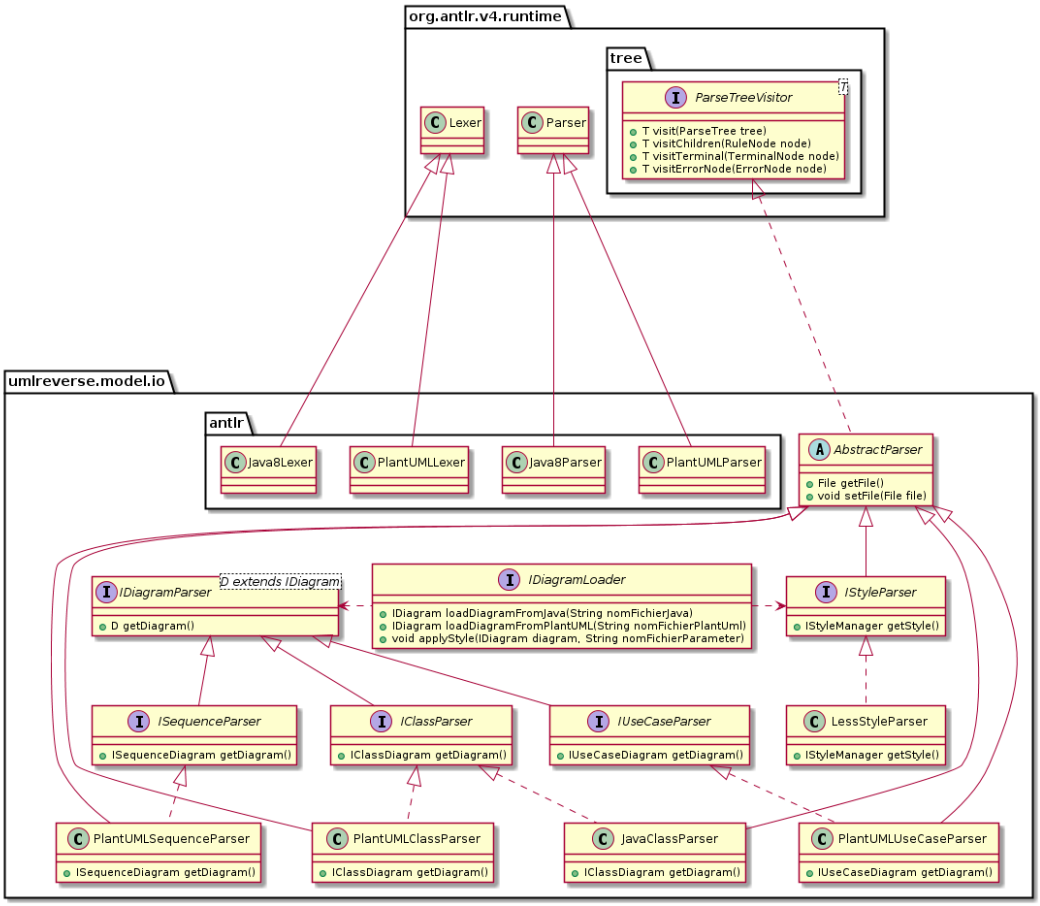
5.2.3 Visiteur



5.3 IDiagramLoader

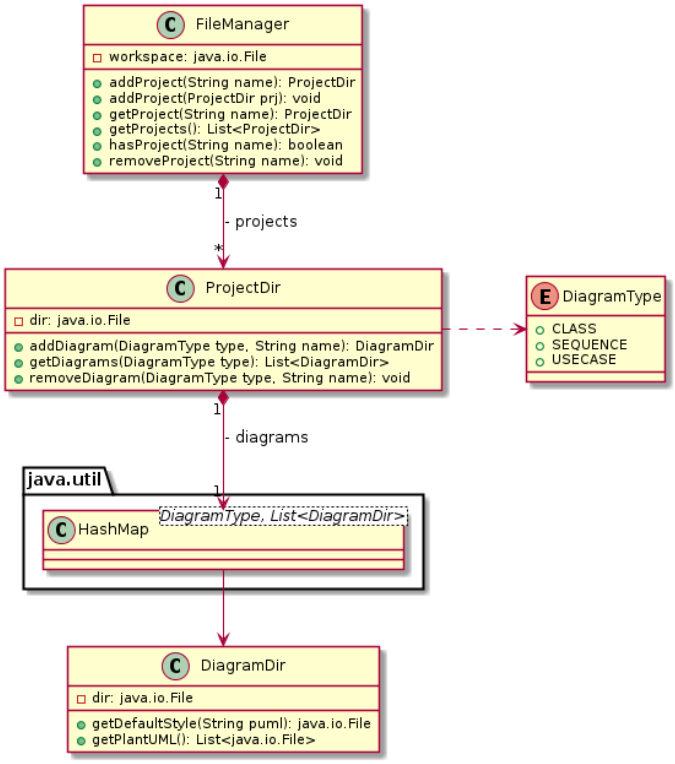
Il s'agit de la classe servant de façade à toutes les fonctionnalités qui nécessitent de parser avec *antlr* un fichier. Cette classe permet de créer un diagramme à partir d'un fichier java ou plantuml. Il permet d'appliquer un fichier de style à un diagramme. C'est cette classe qui utilise *Dot* pour placer initialiser les diagrammes avec un placement automatique.

Diagramme de classes du paquet umlreverse.model.io



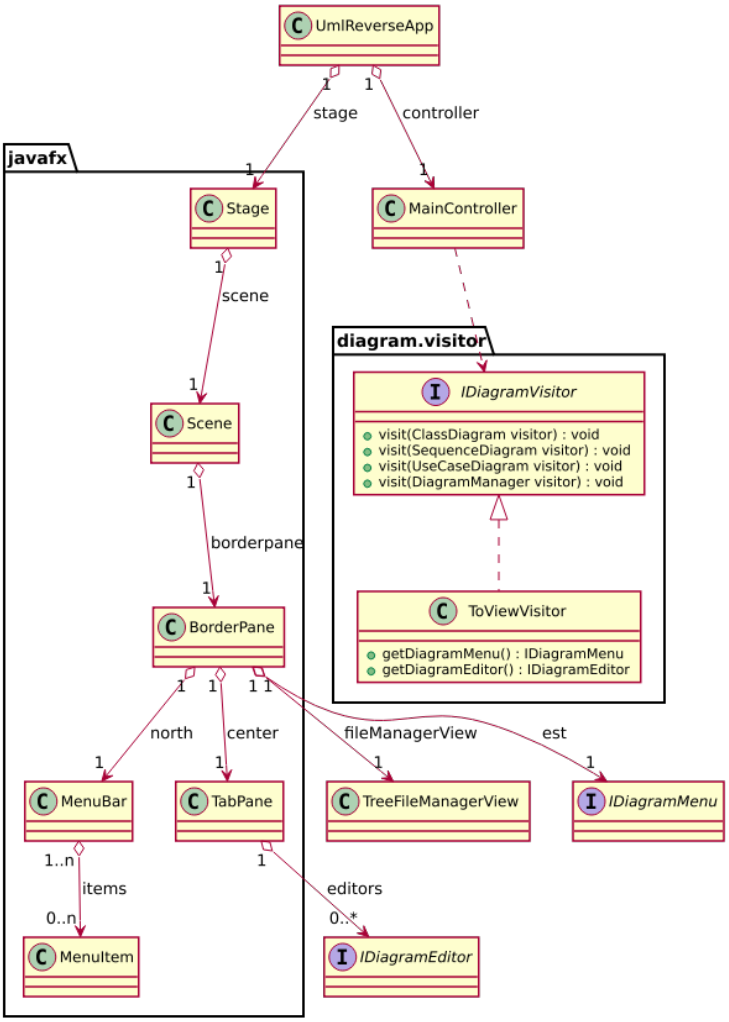
5.4 filemanager

Ce package contient les classes qui gèrent les projets et les différents fichiers contenus. C'est la classe *FileManager* qui sert de modèle pour la partie gauche qui sera définit ensuite.



6 Paquetage ui

L'application s'appuie sur la structure JavaFX avec la classe applicative UmlReverseApp qui est composée d'un stage qui en suivant la hiérarchie JavaFX nous ammène au BorderPane. Le BorderPane nous servira de base pour les différents éléments de notre application, tel qu'une TreeFileManagerView, une MenuBar, un IDiagramEditor et un IDiagramMenu.

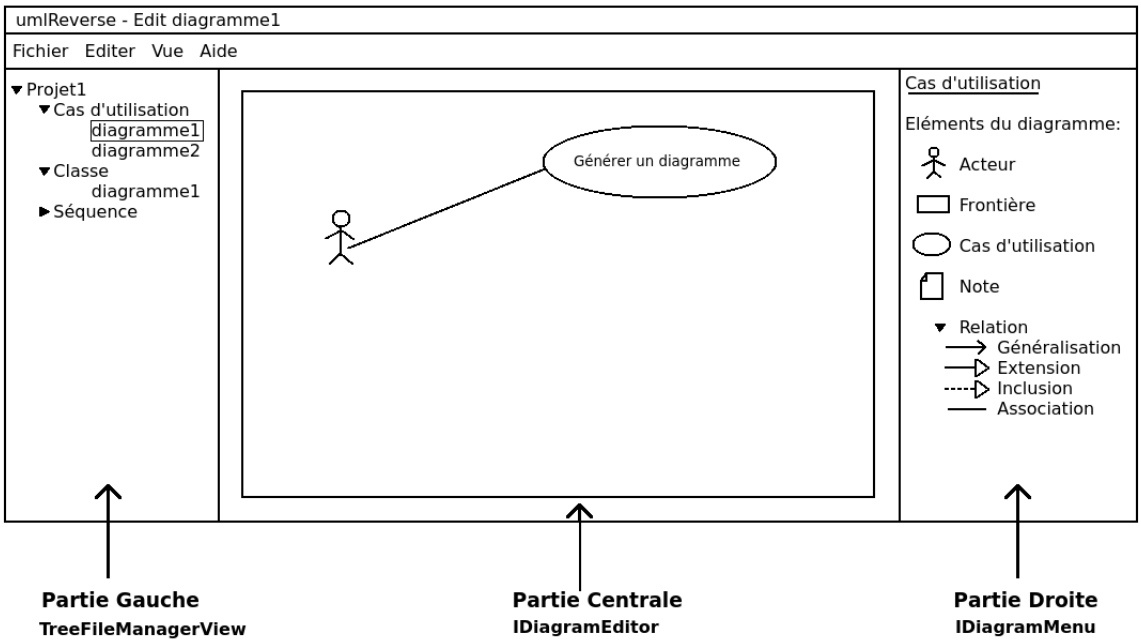


Ce paquetage contient toutes les classes utilisées pour gérer la vue. Ces classes sont toujours associées à un contrôleur. Les vues sont codées en fxml grâce à un logiciel de construction de fxml (SceneBuilder). Chaque contrôleur aura le même nom que la vue associée avec le mot "Controller" ajouté à la fin. ui contient 2 paquetages :

— view : Contient les différentes vues intégrées dans l'application. Le paquetage contient également tous les contrôleurs associés à leur vue.

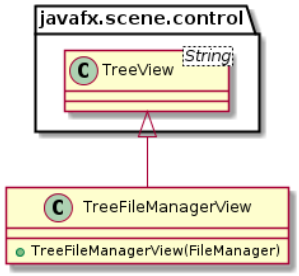
— component : Contient toutes les classes utilisées pour construire les vues. Ce sont leurs composants.

L'application graphique est l'association de plusieurs vues dans un BorderPane.



6.1 La partie gauche

La partie gauche présente la liste des projets disponibles et des diagrammes qu'ils contiennent, et permet de naviguer entre eux. Cette tâche est accomplie par un modèle qui gère les fichiers, accompagné d'une vue hiérarchique constructible à partir de ce modèle.



6.2 La partie centrale

La partie centrale sert à éditer graphiquement un diagramme. La vue principale de cette partie est codée en fxml : DiagramEditor.fxml. Et associé à son contrôleur DiagramEditorController.java.

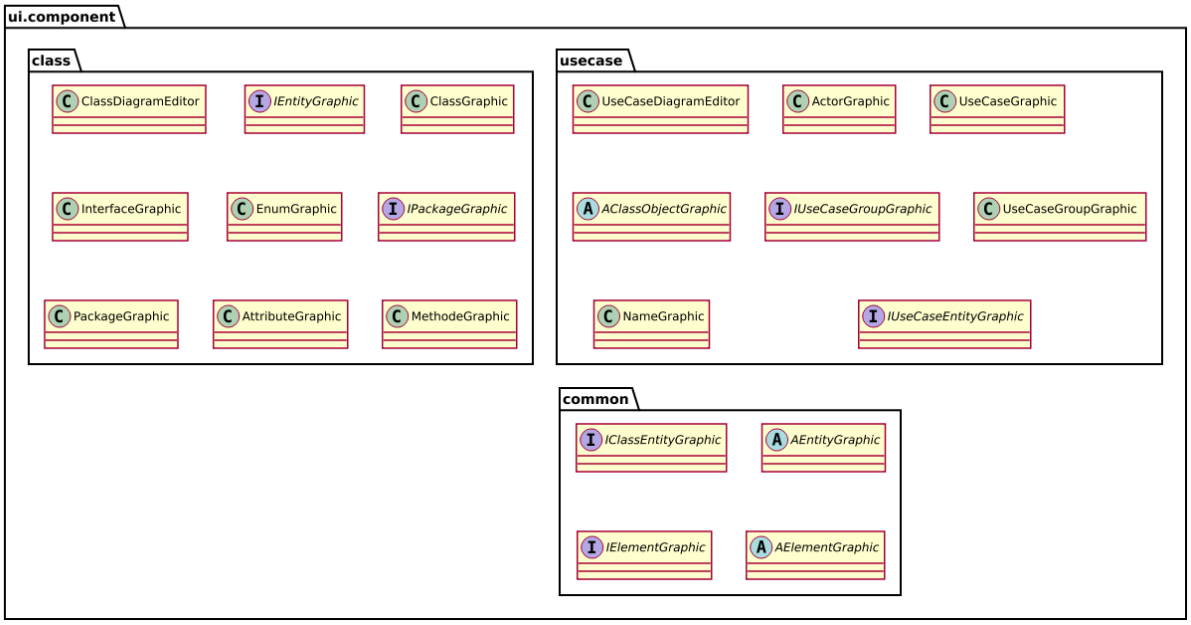
Explication : Toutes les classes qui finissent par Graphic sont des réprésentations graphiques des élements du modèle. Elles contiennent des écouteurs de souris sur elles mêmes pour pouvoir permettre aux utilisateurs de les modifier ce qui modifiera le modèle directement. La modification du modèle modifie obligatoirement la vue du diagramme (les éléments graphiques donc) grâce à des écouteurs sur le modèle.

6.2.1 Editeur de diagramme

La partie MVC a été volontairement omise pour éviter de surcharger le diagramme de classe. Elle sera par contre implémenté dans les parties prévues.
Dans les diagrammes ci contre les contrôleurs controleront les actions de IDiagramEditor.
Nous appelons les éditeur de diagrammes des IDiagramEditor. Nous en avons 2 différents, un pour chaque type de diagramme à éditer.

Paquetages

Les entités



Les relations

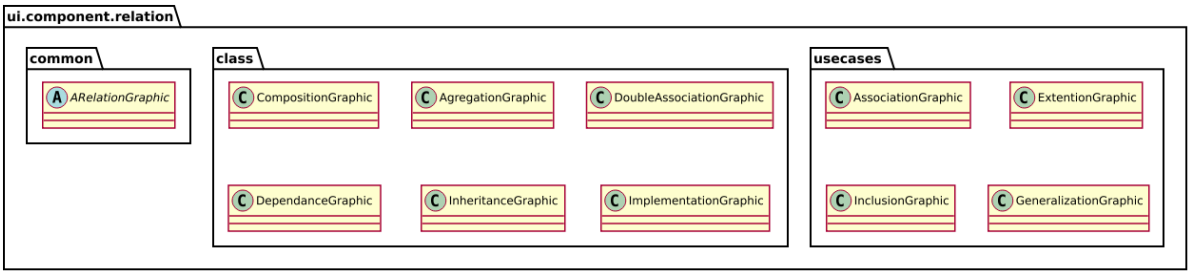
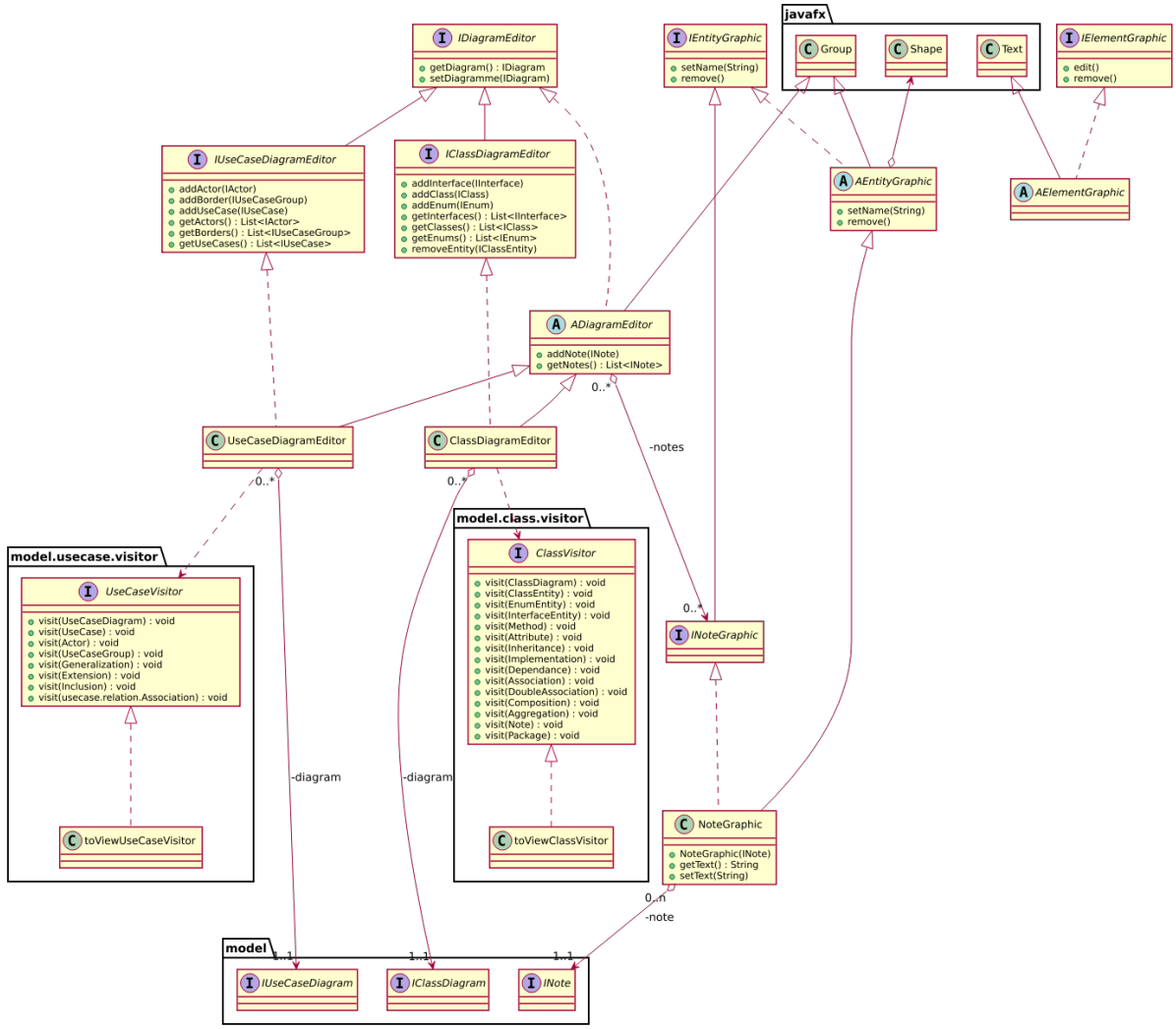


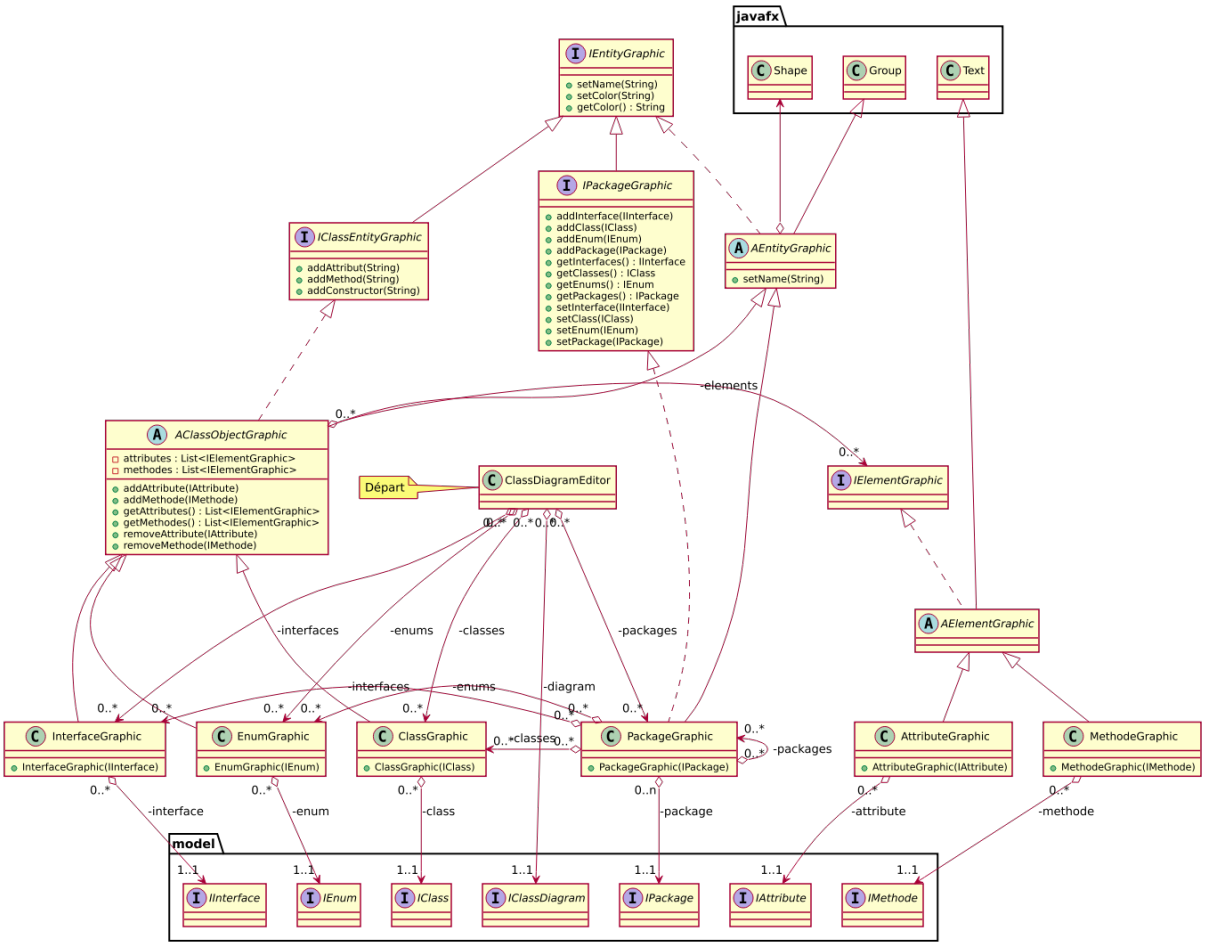
Diagramme de classe



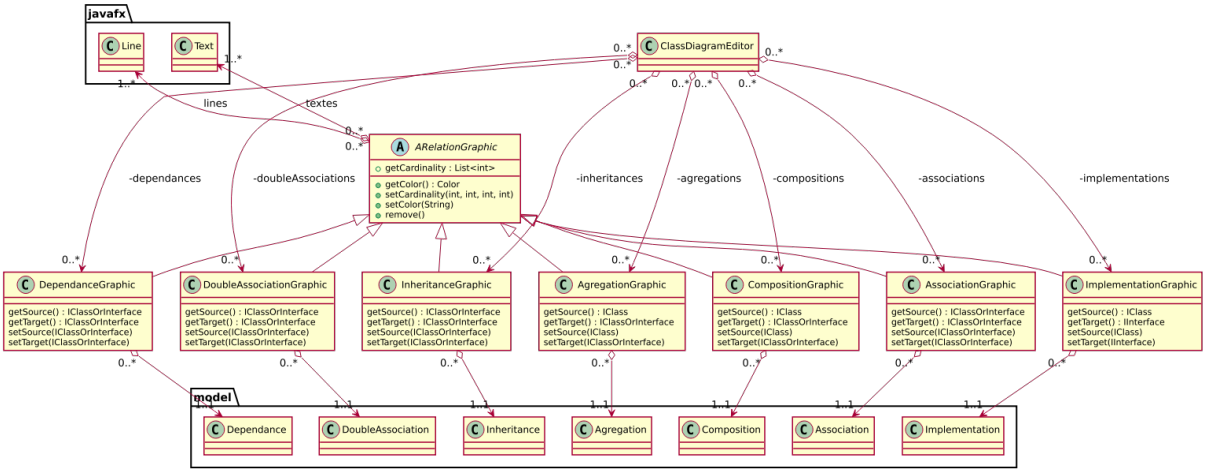
Dans ce diagramme il se trouve les deux DiagramEditor prévus pour ce produit avec les classes qu'elles ont en commun. Les visiteurs de ce diagramme servent à dessiner la vue.

6.2.2 Editeur de diagramme de classe

Gestion des entités

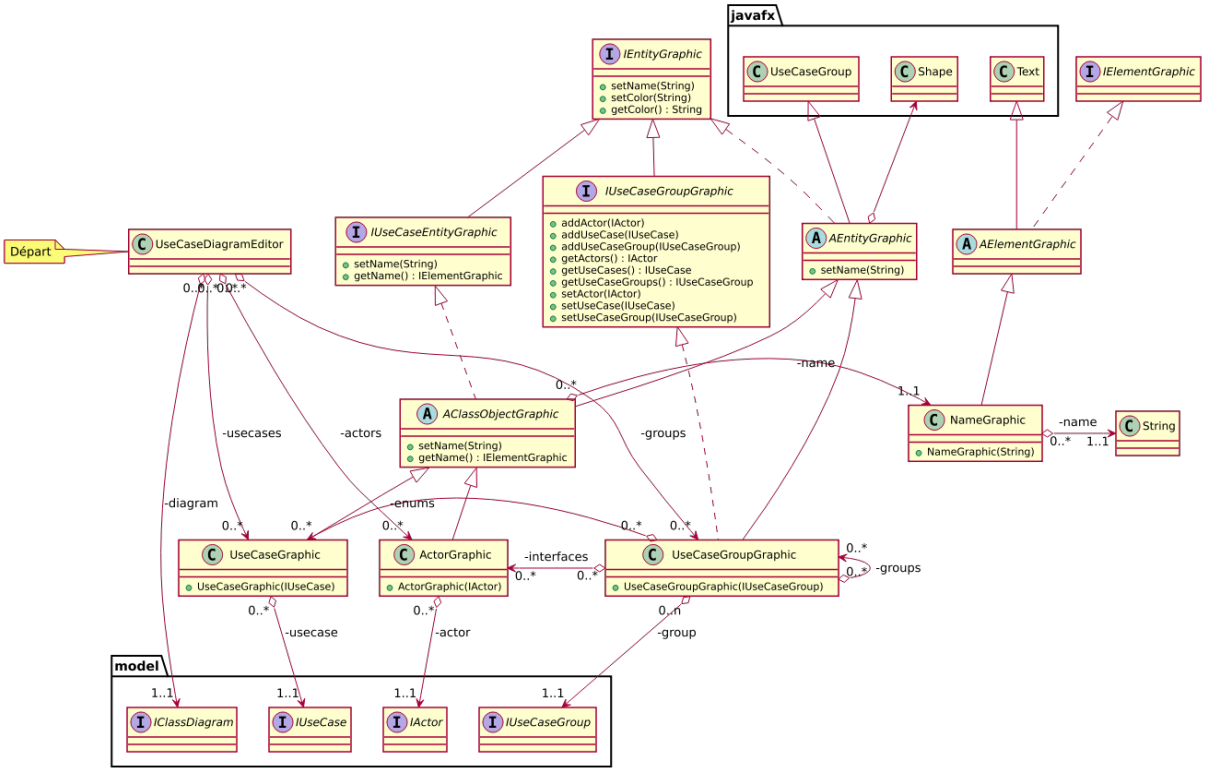


Gestion des relations

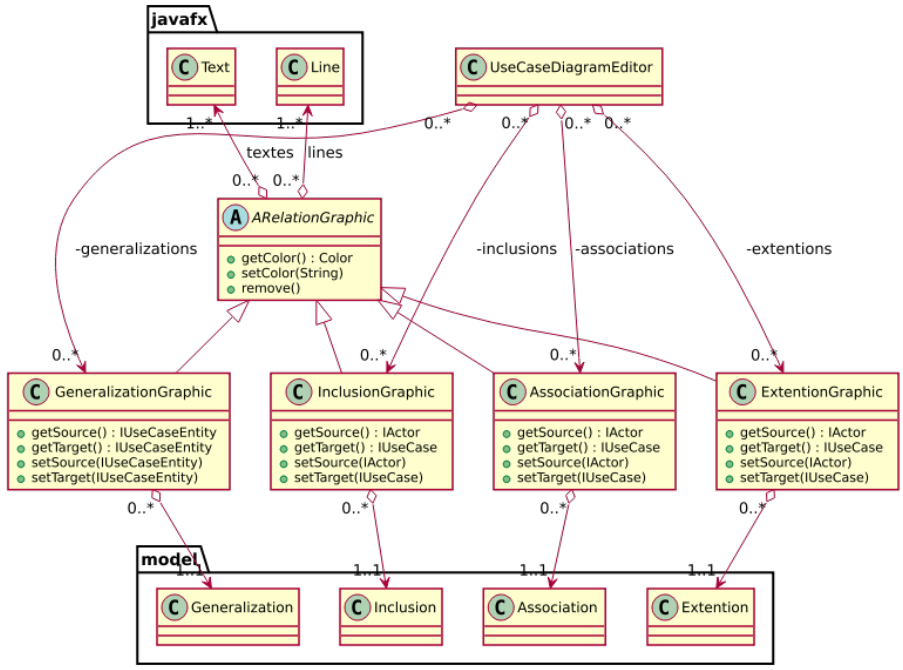


6.2.3 Editeur de diagramme de cas d'utilisation

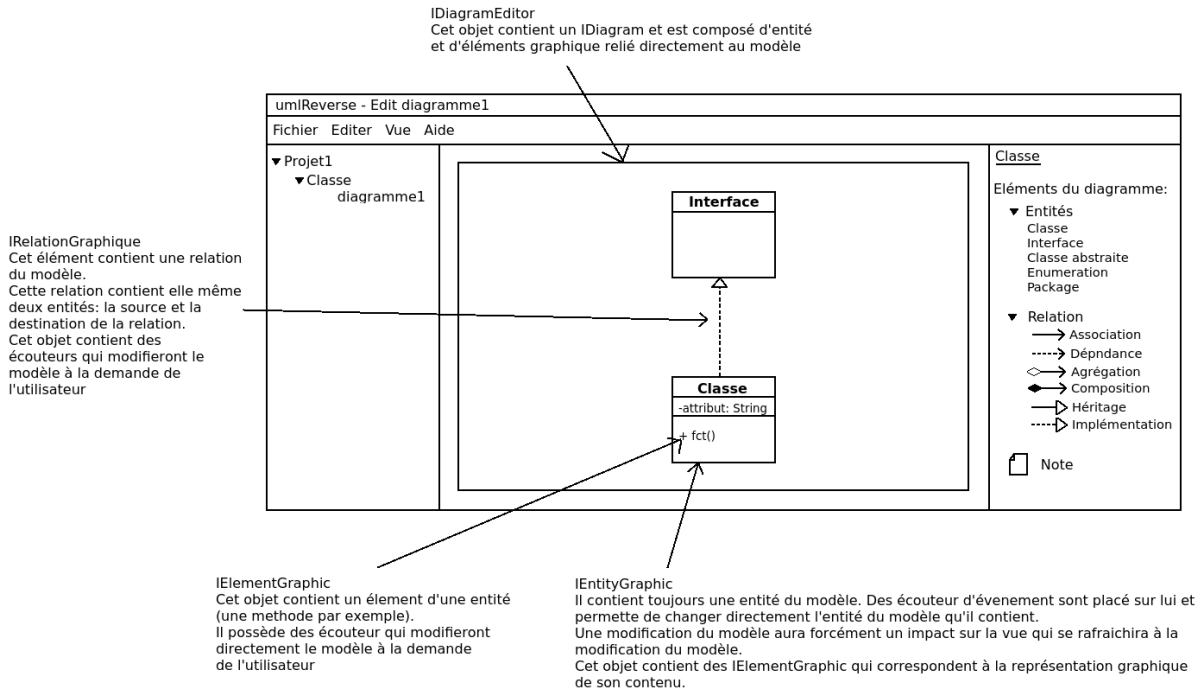
Gestion des entités



Gestion des relations



6.2.4 Explications

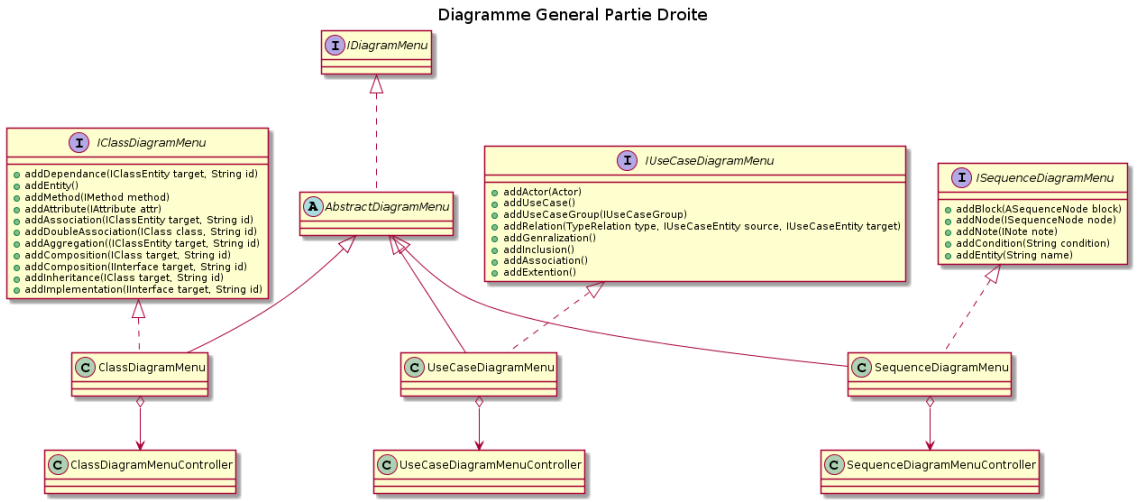


6.3 La partie droite

6.3.1 Déroulement

Quand un utilisateur séléctionne un diagramme l'action fait appel au contrôleur qui va passer par l'interface IDiagram du modèle et le visiteur permet de déterminer le type de diagramme selectionné. L'interface IDiagramMenu nous permet d'ajouter des entités et des relations dans un diagramme grâce au contrôleur qui gère les actions.

6.3.2 Diagramme de classe

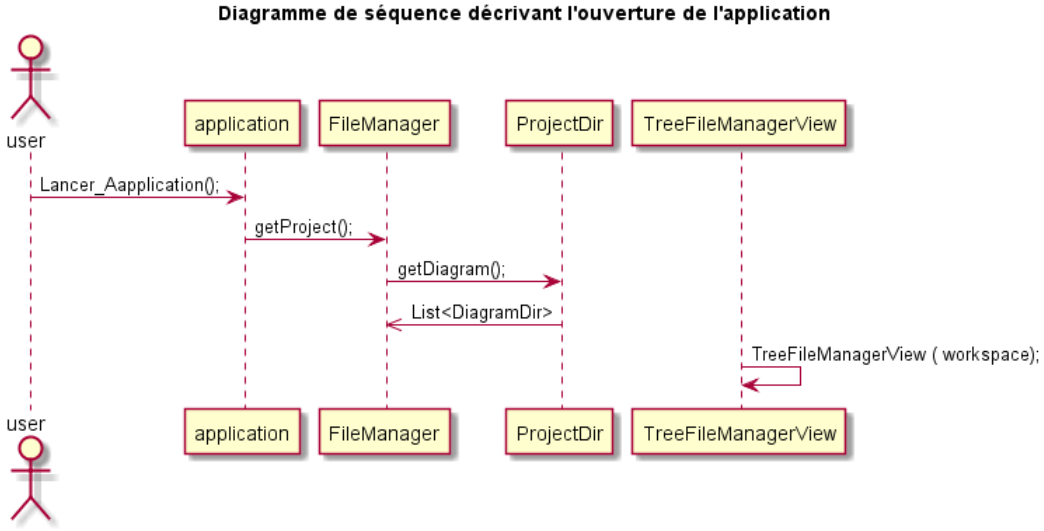


7 Paquetage main

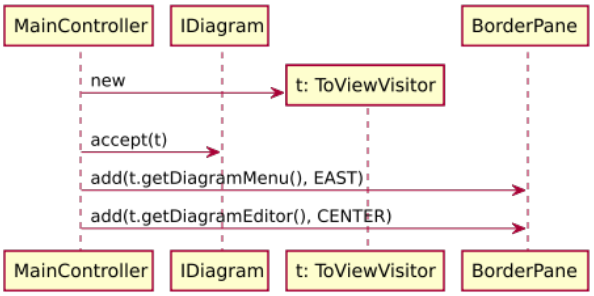
Contient la classe qui lance l'application. C'est le point d'entrée. Il charge les différentes vues du paquetage ui.view en les rassemblant toutes dans un BorderPanel.

8 Diagramme de Séquence

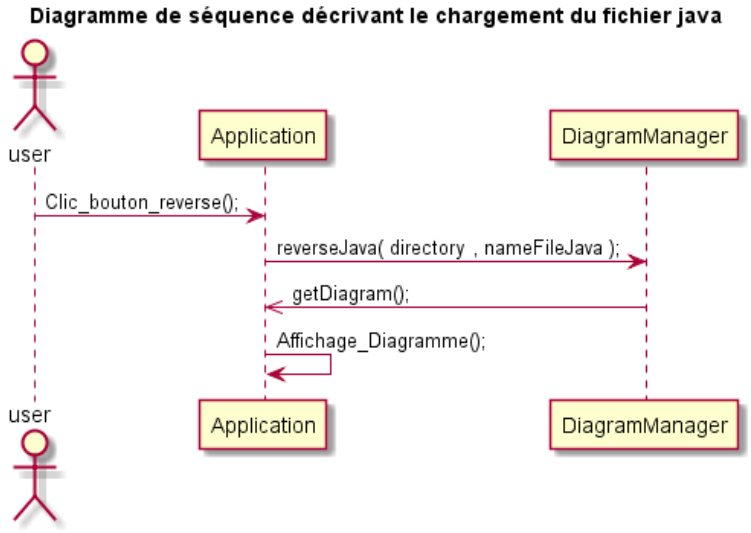
8.1 Ouverture de l'application



8.2 Chargement d'un diagramme dans la vue



8.3 Chargement d'un fichier java



9 Extensibilité

L'architecture a été pensé de façon à vérifier l'exigence EXF_70 (code modulaire, ajout de nouvelles fonctionnalités possible dans le code).

Modèle Le modèle a été pensé à facilement ajouter de nouveau type de diagramme. Pour ce faire le gestionnaire de diagramme ne se soucis pas du type de diagramme. La plupart des actions demandées sur un diagramme de manière extérieur au modèle se font par le biais des visiteurs. Pour ajouter un nouveau type de diagramme il suffit de créer une nouvelle classe qui implémente IDiagram et de mettre à jour les visiteurs globaux.

Partie gauche de l'application Les différentes classes créées peuvent être héritées afin d'étendre leurs fonctionnalitées ainsi que d'ajouter de nouveaux types de diagramme. L'architecture des dossiers d'un projet a été pensée afin que l'ajout de nouveau type de fichier pour la sauvegarde des diagrammes soit simple sans rendre les projets de cette version obsolète.

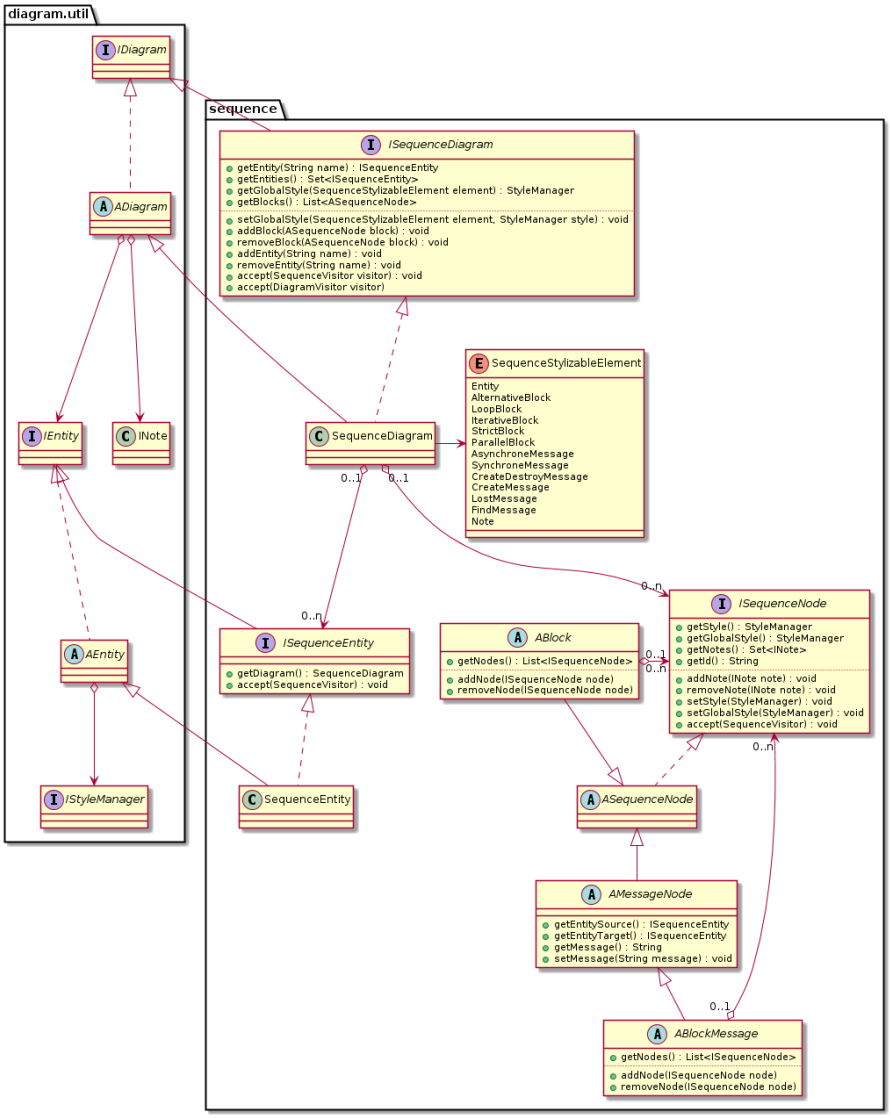
IDiagramEditor Tout d'abord, l'ajout de nouveau type de diagramme a été pensé de façon à être possible et facile à implémenter. Pour se faire, il suffit d'ajouter dans le IDiagramEditor une nouvelle classe XXXDiagramEditor. Cette nouvelle classe peut hériter ADiagramEditor si le nouveau type de diagramme peut posséder des notes. L'implémentation de celle ci sera déjà implémentée. Les classes présentes dans le paquetage view.component.common sont des classes qui peuvent être utilisées dans n'importe qu'elle type de diagramme. Ce sont généralement des classes abstraites qui font une partie du travail. Ce qui évite de tous refaire à chaque fois.

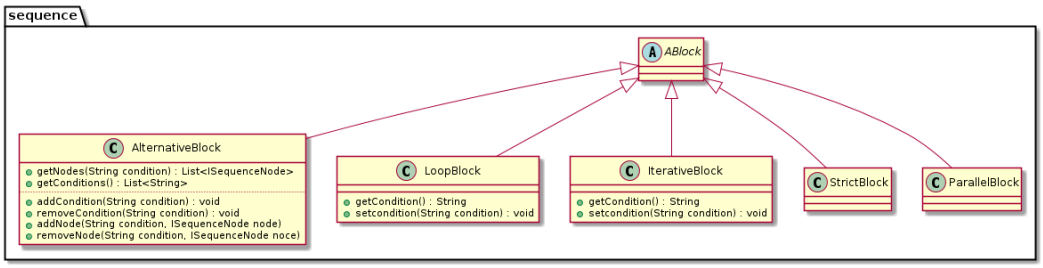
IDiagramMenu Cette partie peut aussi facilement ajouter des nouveaux types de diagrammes. Il suffit de rajouter un nouveau type de menu et d'implémenter la bonne interface.

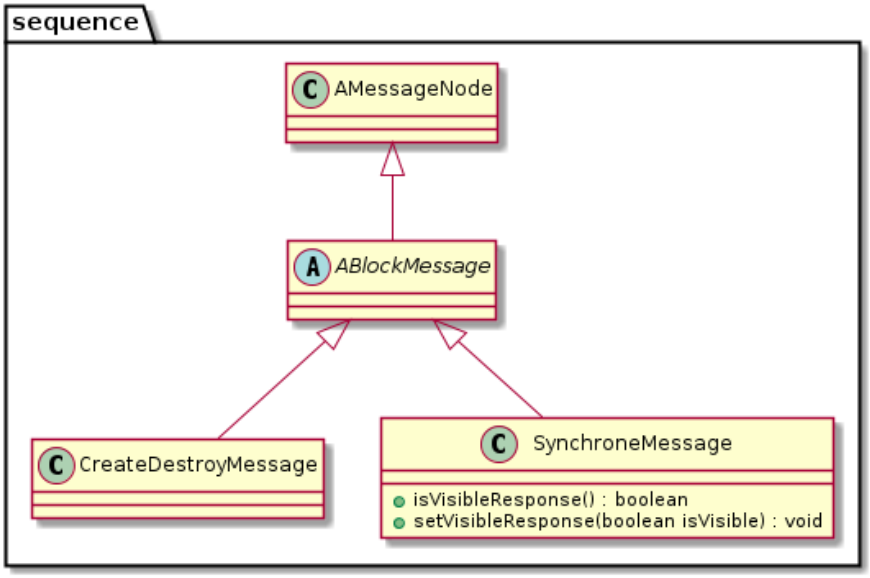
10 Annexe

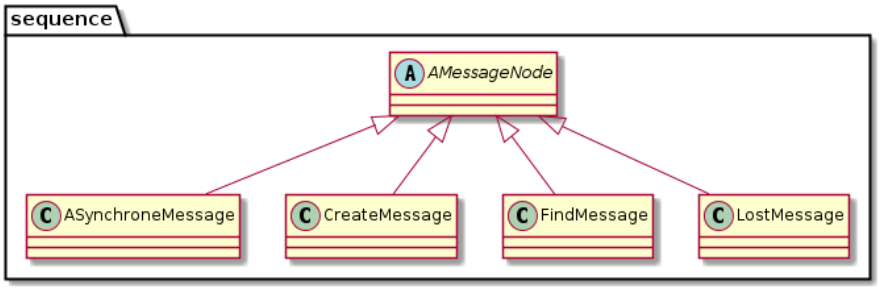
10.1 Modèle du diagramme de séquence

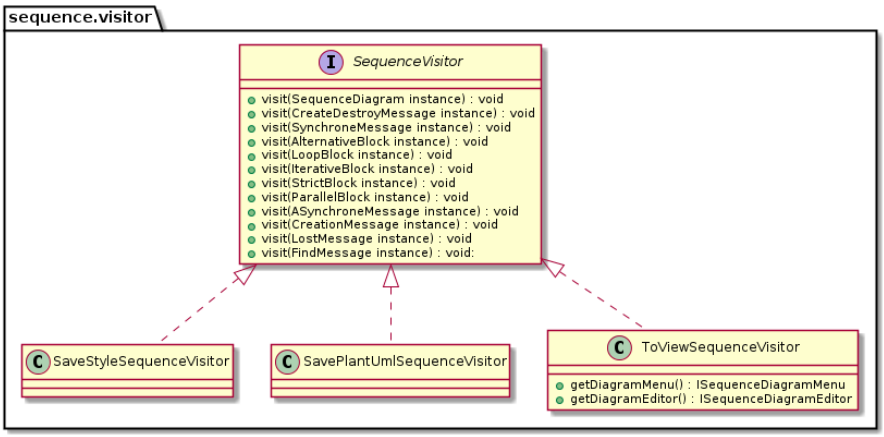
Les diagrammes suivants ont été pensés au cas où nous arrivions à finir le projet avec de l'avance.






sequence
AMessageNode
ABlockMessage
CreateDestroyMessage
SynchroneMessage
isVisibleResponse() : boolean
setVisibleResponse(boolean isVisible) : void


sequence
AMessageNode
ASynchroneMessage
CreateMessage
FindMessage
LostMessage


sequence.visitor
SequenceVisitor
visit(SequenceDiagram instance) : void
visit(CreateDestroyMessage instance) : void
visit(SynchroneMessage instance) : void
visit(AlternativeBlock instance) : void
visit(LoopBlock instance) : void
visit(IterativeBlock instance) : void
visit(StrictBlock instance) : void
visit(ParallelBlock instance) : void
visit(ASynchroneMessage instance) : void
visit(CreationMessage instance) : void
visit(LostMessage instance) : void
visit(FindMessage instance) : void:
SaveStyleSequenceVisitor
SavePlantUmlSequenceVisitor
ToViewSequenceVisitor
getDiagramMenu() : ISequenceDiagramMenu
getDiagramEditor() : ISequenceDiagramEditor