heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

Éditeur de diagramme de classes UML

Projet de diplôme 2010 – 2011

Nom du projet	Slyum
Étudiant	Miserez David
Responsable	Donini Pier
Département	TIC
Filière	IL
Date	29 juillet 2011

Cahier des charges

Le but de ce travail de diplôme est de concevoir un logiciel permettant la création de diagrammes de classes de façon rapide et intuitive. Pour pouvoir être utilisé facilement dans un cadre scolaire.

L'étudiant devra dans un premier temps étudier les diagrammes de classes et leur fonctionnement. Il devra également étudier le langage XML ainsi que XML-Schema afin de pouvoir exporter et importer des fichiers de ce type pour représenter un diagramme de classes. L'apprentissage de Swing est également requis.

Un meta-schema de diagrammes de classes ainsi que la structure du système sera représentée à l'aide de diagrammes de classes UML. Il doit y avoir un premier diagramme de classes pour la représentation des diagrammes de classes (modèle). Un second pour la représentation graphique des diagrammes de classes (vue) et un troisième pour la structure des composants Swing de l'application (vue).

Une fois la conception terminée l'étudiant devra coder en Java l'application et ses fonctionnalités.

Un journal de travail doit être maintenu à jour tout au long du projet. Un rapport intermédiaire devra être fournit à la moitié du projet (17 juin 2011) et le rapport final à la fin (29 juillet 2011).

La défense du projet de diplôme se déroulera entre le 5 et 16 septembre 2011 (semaines 36 et 37). Une présentation lors des portes ouvertes sera également nécessaire. Cette présentation requiert la création d'une affiche (à rendre avant le 1 septembre 2011) et sera faite le 7 octobre 2011.

Le programme offrira les fonctionnalités suivantes :

- Création de classes, d'interfaces et de classes internes
- Création d'attributs et de méthodes
- Création d'associations (binaire, n-aire, composition, agrégation, classe d'association).
- Création de relations d'héritages (généralisation, réalisation).
- Création de relations de dépendances.
- Exportation du diagramme de classes au format XML.
- Il doit être possible d'enregistrer et d'ouvrir un projet dans un fichier pour pouvoir le reprendre par la suite.
- Les diagrammes de classes doivent pouvoir être exporté dans un format graphique (GIF, JPG, PNG).
- Imprimer les diagrammes de classes directement depuis l'application.

Table des matières

Résumé (Summary)

Français

Il existe dans le commerce et dans le monde open source de nombreux éditeurs de diagrammes de classes UML. Certains d'entre eux offrent de nombreuses fonctionnalités, mais d'une part l'aspect visuel du diagramme est trop souvent négligé et, d'autre part, l'ajout d'éléments dans le diagramme est peu confortable pour l'utilisateur.

Le but de ce projet est donc de concevoir un éditeur de diagrammes de classes simple d'utilisation, se conformant à la norme UML 1.4.

English

A lot of diagrams editor exist in industry and open source world. But mainly of them are complicate to use and not very pleasant to see. Moreover adding new elements are difficult for user.

That’s why we will develop this project. This application will be easy to use and only useful elements will be integrated in. The purpose of this project is it can be used in UML learning.

1 Introduction

Pour ce travail de diplôme, le but est de concevoir une application permettant la création de diagrammes de classes UML. Les diagrammes de classes permettent, en génie logiciel, de représenter graphiquement la structure d'un système à l'aide d'éléments structurels (tel que les classes et d'interfaces) ainsi que les relations entre ceux-ci. Ils sont utilisés lors de la conception d'applications orientées objets.

Ce projet se distingue des autres applications déjà présentes sur le marché (qu'elles soient open source ou commerciales) par une interface simple et accessible. Il doit pouvoir être pris en main rapidement et permettre de concevoir des diagrammes de classes avec facilité dans le but d'être utilisé dans l'apprentissage de la modélisation UML. Pour ce faire, seuls les éléments essentiels et indispensables à la création de diagramme de classes sont présents. Les actions non conforme à la POO sont également proscrites (une classe qui est à la fois parente et enfant d'une même classe par exemple).

1.1 Informations techniques

Nom du projet	Slyum
Langage de programmation	Java
Bibliothèque graphique	Swing
Format d'enregistrement	XML
Version UML	UML 1.4
License	GNU GPL v3
Environnement	Académique
Participant	Miserez David
Responsable	Pier Donini
Gestionnaire de version	Subversion
Lien internet	http://code.google.com/p/slyum/
Dernière version	1.0

1.2 Structure du rapport

La première partie (chapitre 2) du rapport se concentre sur le langage UML utilisé dans la conception de diagrammes de classes. Elle décrit les notations et représentations utilisées ainsi que les contraintes et limitations qui seront inclues dans le projet.

La seconde partie (chapitre 3) argumente le choix de la bibliothèque graphique ainsi que la structure générale du projet (chapitre 4).

La troisième partie (chapitre 5) est une analyse du meta-schema construit pour représenter les diagrammes de classes de l'application.

La quatrième partie (chapitre 6) détaille la structure pour la représentation graphique du diagramme de classes ainsi que la structure des composants Swing (interface utilisateur) de l'application (chapitre 7).

La dernière partie (chapitre 8) traite le langage utilisée pour l'exportation et l'enregistrement des diagrammes de classes (XML) ainsi que la description utilisée (XML-Schema).

Information :

- *Lorsqu'un chapitre fait référence à une classe ou à un paquetage de la structure du projet, le nom dudit chapitre est suivi du nom de la classe ou du paquetage entre parenthèses.*
- *Souvent, les diagrammes de classes présentés dans ce rapport ne contiennent pas toutes les classes, attributs, ni méthodes qu'ils possèdent en réalité. Ce choix permet une meilleure lisibilité des diagrammes en ne mettant que les éléments le plus importants pour la compréhension de la structure du système.*

2 Unified Modeling Language

UML (Unified Modeling Language) permet, en génie logiciel, de concevoir la structure d'un projet orienté objet sous forme de diagramme. UML définit treize types différents de diagrammes dont les diagrammes de classes qui sont traités ici. Ce chapitre explique comment les différents éléments UML sont intégrés et représentés dans le projet.

2.1 Structures de données

Les structures de données (Illustration 1) représentent les classes et les interfaces ainsi que leurs attributs et opérations.

En UML, une structure de donnée est représentée par un rectangle séparé verticalement en trois parties. La première partie contient le nom de la structure, la seconde ses attributs et la dernière ses opérations.

Remarques sur les structures de données :

- Une interface est différenciée d'une classe par son stéréotype « Interface ». Le stéréotype est inscrit au-dessus du nom de la structure.
- Le nom d'une classe abstraite ainsi que le nom des interfaces est écrit en italique.
- Les types peuvent être ajoutés manuellement par l'utilisateur. Les types primitifs sont déjà intégrés dans le programme.

2.1.1 Opérations

Les opérations sont représentées dans la dernière partie d'une structure de données sous forme textuelle. Elle commence par un caractère représentant la visibilité, puis le nom de la méthode. Ensuite vient, entre parenthèses, sa liste de paramètres. Chaque paramètre est séparé par une virgule et s'écrit de la manière suivante : le nom, suivit de deux points (« : ») puis le type. Après les paramètres se trouve le type de retour, précédé du caractère deux points (« : »).

Grammaire des opérations :

Opération :

```
(("+"|"-"|"#"|"~")Nom"("{Paramètres}"):" Type_De_Retour
```

Paramètre :

```
(Nom":"Type)|(Nom":"Type", ")
```

Exemple : `+setSize(width:int, height:int):void`

Remarques sur les opérations :

- Une méthode statique est soulignée.
- Une méthode abstraite est écrite en italique.

2.1.2 Attributs

Les attributs sont écrits dans la seconde partie de la structure de données. L'attribut commence par un caractère indiquant sa visibilit, puis son nom suivit de deux point (« : ») et enfin le type.

Grammaire des attributs :

```
("+"|"-"|"#"|"~")Nom":"type["["[taille]"]"]
```

Exemple : `-size : Dimension`
`-arraySize : Dimension[]`

Remarque sur les attributs :

- Un attribut constant est représenté un ajoutant le mot-clé *{const}* après sa signature.

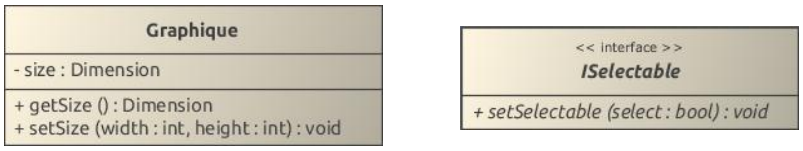


Illustration 1 Représentation UML d'une classe et d'une interface (Slyum)

2.2 Relations

Les relations en UML représentent comment les structures de données interagissent entres-elles. Ce chapitre expose les différents types de relations et leur représentation graphique.

2.2.1 Héritage

La représentation d'héritage se fait à l'aide d'une flèche partant de l'enfant et pointant sur le parent (Illustration 2). Une flèche trait-tillé représente une réalisation, une ligne droite une généralisation. Dans Slyum, il n'y a qu'un seul type de relation d'héritage qui s'ajustera en fonction de la classe parente.

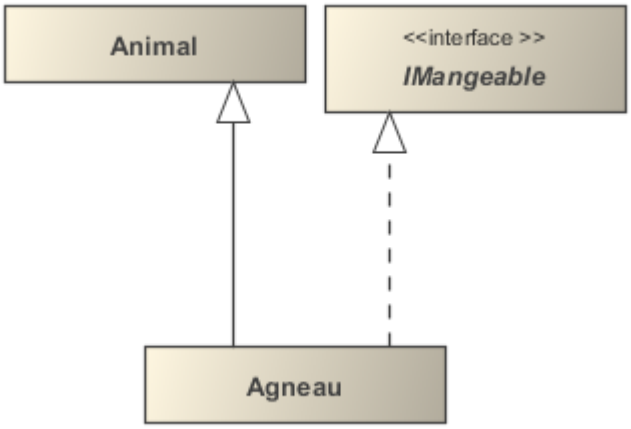


Illustration 2 Représentation des relations d'héritages (Slyum)

2.2.2 Dépendances

Les dépendances sont représentées à l'aide d'une flèche ouverte trait-tillé (Illustration 3) et peuvent posséder un label donnant plus d'informations sur le rôle de la dépendance.

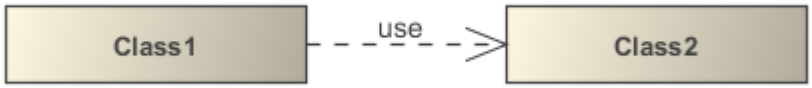


Illustration 3 Représentation d'une relation de dépendance (Slyum)

2.3 Association

Il y a différents types d'associations : binaire, n-aire, agrégation, composition et classes d'association. L'illustration suivante indique un prototype de la manière dont ces différentes associations sont représentées dans l'application.

Représentation de la relation d'association binaire. Une association binaire peut être dirigée en lui ajoutant une flèche, ou bidirectionnelle si elle n'a pas de flèche. Une association binaire comprend deux classes ; une classe source et une classe cible. Dans le cas d'une association unidirectionnelle (dirigée), la classe source pointe sur la classe cible.

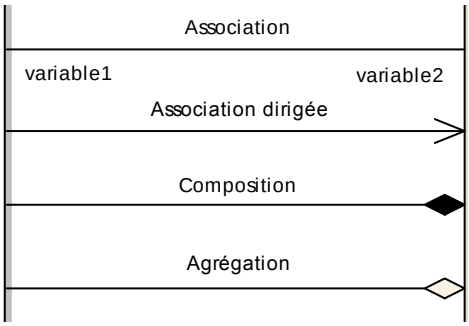


Illustration 4 Représentation des associations (Enterprise Architect)

Les associations n-aire (ou multi-association) sont représentées (Illustration 5) à l'aide d'un losange reliant les structures de données impliquées dans l'association. Les associations n-aire ne peuvent ni être dirigées, ni être des compositions ou des agrégations.

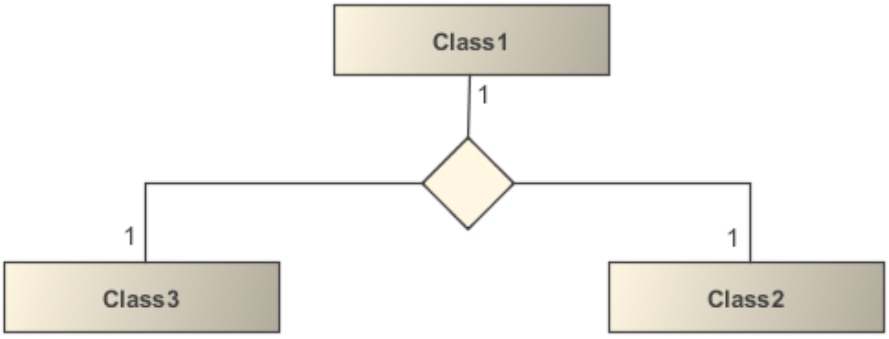


Illustration 5 Association n-aire (Slyum)

Les classes d'associations sont représentées avec une classe reliée à une association par une ligne trait-tillée (Illustration 6).

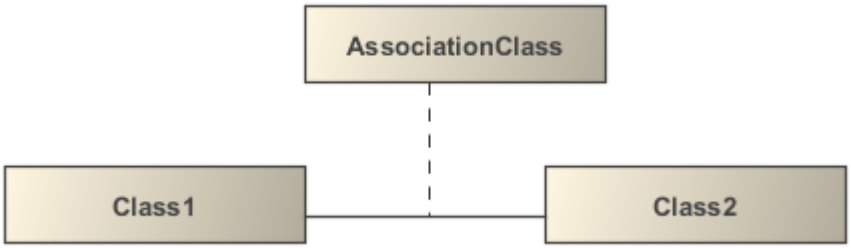


Illustration 6 Représentation d'une classe d'association (Slyum)

Une croix dans un cercle à la place d'une flèche représente une classe interne (Illustration 7). Les classes internes n'ont pas encore été implémentées.

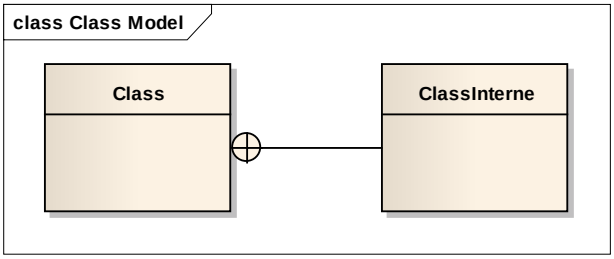


Illustration 7 Représentation d'une classe interne (Enterprise Architect)

2.3.1 Nom des associations

Toutes les associations peuvent posséder, facultativement, un nom qui s'affiche sous forme d'un label aux alentours de l'association. Le nom d'une association définit souvent l'action que joue l'association entre les deux classes. Elle peut être précédée ou suivie du caractère '<' ou '>' pour indiquer le sens de lecture. Syntaxiquement, le nom d'une association est du texte, il accepte n'importe quel caractère.

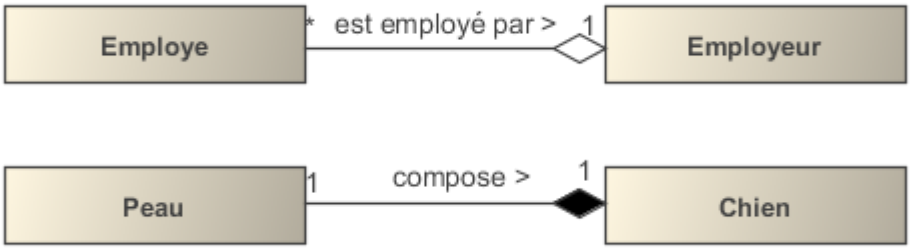


Illustration 8 Représentation de noms d'associations (Slyum)

2.3.2 Rôles des associations

Le rôle d'une association indique des informations supplémentaires sur le rôle joué par une classe avec l'association. Concrètement, les rôles d'une association indiquent le nom et la visibilité (public, private, protected ou package) de l'attribut qui sera créé dans les classes participantes de l'association.

Les rôles sont affichés aux extrémités de l'association avec la syntaxe suivante (Illustration 9) :

`('+' | '-' | '~' | '#')␣{Nom_Du_Rôle}`

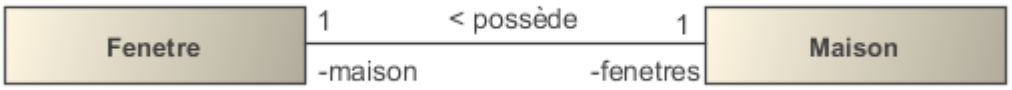


Illustration 9 Affichage des rôles d'une association (Slyum)

Dans cet exemple, il y a deux rôles « maison » et « fenetres » ayant une visibilité « private ». Le nom d'un rôle suit la syntaxe d'un attribut quelconque, il ne peut pas avoir d'espaces, ni de caractères spéciaux (excepté '_'), ni commencer par un nombre.

2.3.3 Multiplicités des associations

Les rôles possèdent également une multiplicité. La multiplicité d'un rôle indique le nombre d'occurrences (de références) qu'une classe possède sur une autre classe de l'association. Par défaut cette multiplicité est de 1. Dans Slyum, les multiplicités les plus utilisées sont disponibles par défaut (Illustration 10). Une multiplicité correspond à deux bornes ; une inférieure et une supérieure, pouvant être identiques. La borne inférieure ne peut pas être supérieure à la borne supérieure.

Syntaxe d'une multiplicité :

`{Borne_Inférieure}'..'{Borne_Supérieure}`

Si les deux bornes sont identiques, la multiplicité est représentée par :

`{Un_Nombre}`

Pour finir, on utilise le caractère '*' (étoile) pour indiquer qu'une borne est « à l'infinie » (se lit « plusieurs »).

Zéro	0
Zéro ou une	0..1
Zéro ou plusieurs	0 .. * ou *
Une	1
Au moins une	1 .. *

Illustration 10 Multiplicités de base

3 Choix de la bibliothèque graphique

Le choix de la bibliothèque graphique fût fastidieux. Ma première démarche a été de rechercher une bibliothèque déjà existante et proposée gratuitement pour Java. Je me suis dit qu'ainsi, je pourrais me concentrer sur le reste du projet en y ajoutant le maximum de fonctionnalités pour rendre l'utilisation du projet plus simple et assistée.

N'ayant pas trouvé mon bonheur, je me suis tourné vers le toolkit SWT, qui avait l'avantage d'être natif, contrairement à Swing (et donc, j'ai pensé, plus rapide). Après avoir testé plusieurs implémentations différentes d'interfaces utilisateurs et de bibliothèques graphiques, je me suis finalement tourné vers Swing.

3.1 Bibliothèques existantes

Avant de prendre le choix de faire une bibliothèque graphique exprès pour ce projet, j'ai fait différentes recherches de bibliothèques déjà existantes sur internet.

3.1.1 Piccolo2D

Piccolo2D est la plus intéressante bibliothèque graphique en Java que j'ai trouvée pour mon projet. Elle n'est pas trop complexe et surtout elle gère les événements souris / clavier et les zooms. De plus les composants graphiques sont inexistants et doivent être entièrement fait par l'utilisateur, ce qui permet de personnaliser comme bon nous semble la représentation des différents éléments UML.

Le seul problème venait du fait que la bibliothèque, au moment où j'ai commencé le projet, n'était plus mise à jour depuis près de 1 année. Mes dernières vérifications m'informes qu'une nouvelle version vient de sortir.

Site internet : http://www.piccolo2d.org/index.html

3.1.2 JHotDraw 7

JHotDraw 7 est une autre bibliothèque graphique pour Java. Celle-ci a connu plusieurs propriétaires et une foultitude de remaniement. Si bien que les informations sur internet, tout comme les tutoriaux ou manuels sont pratiquement inexistants. De plus, la bibliothèque propose des composants graphiques déjà « tout fait », ce qui ne convenait pas du tout pour les besoins du projet.

Site internet : http://www.randelshofer.ch/oop/jhotdraw/index.html

3.2 SWT (Standard Widget Toolkit)

Ayant vaguement entendu parler de SWT, et ne connaissant pas forcément mieux Swing, je me suis premièrement tourné vers SWT. Le fait que les composants de l'interface graphique soient natifs sur le système d'exploitation sur lequel le programme tourne m'a séduit.

Les premières semaines j'ai commencé par travailler sur l'interface graphique en regardant les compatibilités / incompatibilités inter-système d'exploitation (comme les composants sont natifs, ils ne sont pas toujours gérés, ni affichés de la même manière sur toutes les plateformes).

Ensuite j'ai commencé à développer la bibliothèque graphique. J'avais le choix entre recommencer tout à zéro, en gérant les événements de la souris et les déplacements moi-même. Ou utiliser des composants SWT tels que des canevas. Les canevas sont des composants SWT vides dans lesquels on dessine ce que l'on veut. L'avantage est que les événements SWT sont déjà gérés.

En choisissant la seconde option, je me suis vite heurté à différents problèmes, comme l'impossibilité de rendre un canevas transparent ou l'exportation en image rendue compliquée. Après plusieurs bricolages, j'ai finalement décidé de recommencer en construisant ma propre bibliothèque graphique.

Une fois les bases implémentées, j'ai remarqué que le tout était beaucoup trop lent pour être utilisable correctement sur une machine aux capacités modestes. N'ayant pas trouvé de solution acceptable, je me suis tourné vers Swing, après avoir vérifié que les mêmes actions étaient plus rapides que sur SWT.

Les ralentissements de SWT ne survenaient que sur Windows. Certains parlaient de l'implémentation du double buffer mal conçue, d'autres qu'il n'était même pas activé sur cette dernière plateforme. Au final, sans avoir effectué de trop longues recherches (plus le temps), je me suis tourné sur Swing qui semblait mieux correspondre à mes besoins.

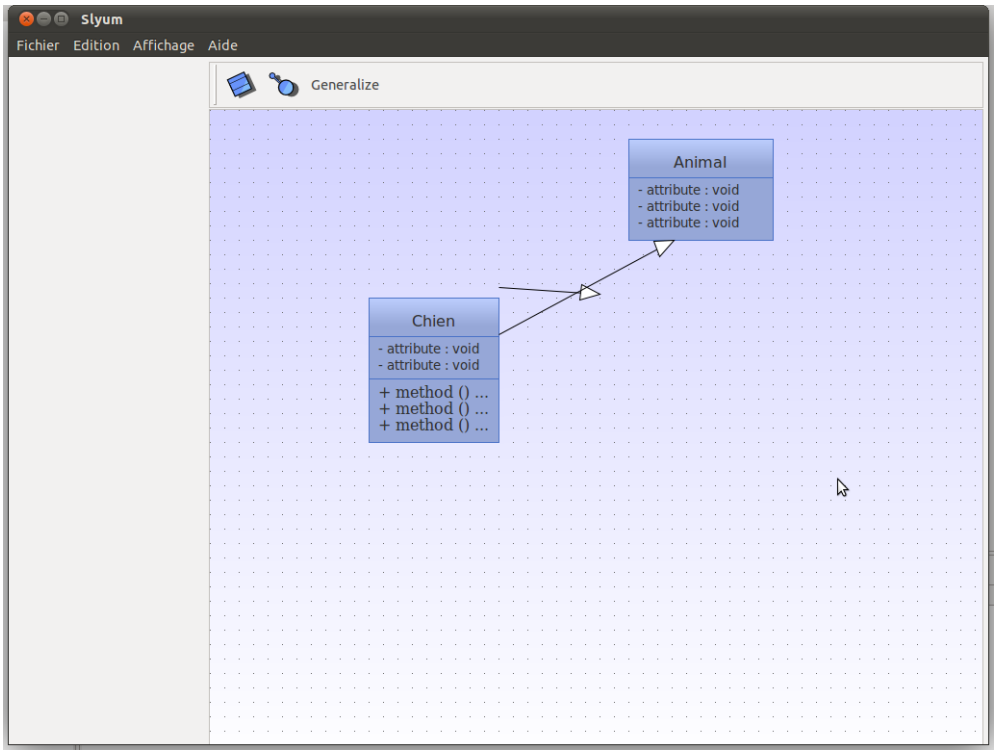


Illustration 11 Capture d'écran du projet sous SWT

3.3 Swing

Pour porter mon projet sur Swing, j'ai tout repris à zéro. J'ai construit une nouvelle structure en tirant parti de mes expériences faites avec SWT. Sans avoir essayé de créer une bibliothèque intégrant des éléments graphiques Swing, je suis parti sur un JPanel dans lequel je dessinais la scène. La structure de base du projet était sensiblement la même que celle de SWT, mais j'ai ajouté un certains nombres de classes pour rendre le projet plus évolutif. En SWT j'avais une seule classe pour le dessin d'une classe, de ses attributs et méthodes, de son nom. J'ai repris le concept pour l'améliorer en ajoutant une classe pour chaque objet graphique du diagramme de classe. La structure de cette nouvelle et dernière approche est présentée au chapitre suivant.

4 Structure générale

Le projet contient quatre parties distinctes (Illustration 12) et utilise une variante du modèle Java « *Observer* ». La première de ces parties représente le modèle (présenté au chapitre 5). Le modèle contient les informations sur la structure du diagramme de classes ; classes, relations, méthodes, attributs, etc.

Les trois autres parties sont des vues. La première des vues est la représentation du diagramme de classes sous forme graphique (présentée au chapitre 6). Les deux autres vues font parties de l'interface utilisateur (Swing), elles permettent d'éditer de manière indirecte le diagramme de classes (présentée au chapitre 7). La première de ces vues est une vue hiérarchique sous forme d'arbre affichant l'arborescence des classes et des relations (chapitre 7.1). L'autre vue est une vue dite de propriétés, cette vue affiche toutes les propriétés des éléments du diagramme de classe (chaputre7.2). Une propriété peut être par exemple de rendre une association dirigée, un attribut statique ou encore une méthode abstraite. Ce sont des propriétés qui ne peuvent pas directement être définie depuis la représentation graphique du diagramme de classe.

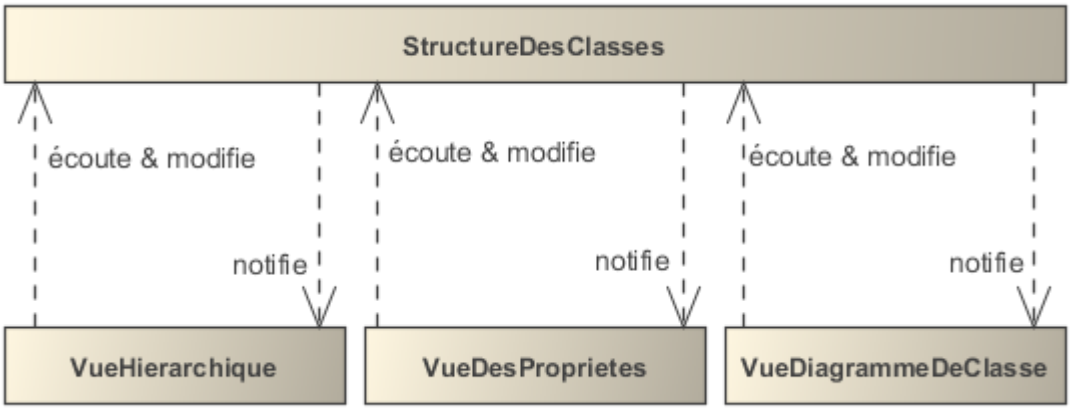


Illustration 12 Structure général du projet

Dans le projet, ces différentes parties peuvent être identifiées par leur nom de paquetage (au sens Java). Il y a trois paquetages principaux :

- **classDiagram** : Correspond au modèle. Contient la structure des classes et leurs relations (héritages, associations, dépendances, etc.).
- **graphic** : Contient les classes permettant de représenter le diagramme de classes graphiquement.
- **swing** : Contient l'interface utilisateur ainsi que la vue hiérarchique et la vue des propriétés.

Chaque sous- paquetage fait l'objet d'un chapitre dédié.

4.1 IComponentObserver

Afin que les différentes vues puissent s'adapter aux changements de la structure des classes, une variante du modèle *Observer* de Java a été créée, appelée IComponentObserver (Illustration 13). Avec un modèle *Observer* ne comportant qu'une seule méthode *update()*, il aurait fallu faire une suite de conditions lors de l'ajout d'un nouvel élément au diagramme de classe. La méthode *update()* ne comportant qu'un paramètre, il aurait fallu passer le nouveau composant créé à la méthode et ensuite chaque vue aurait dû regarder le type du nouveau composant pour créer sa propre instance représentant le composant UML dans la vue donnée (Illustration 14). De plus, comment distinguer la suppression ou l'ajout d'un composant ?

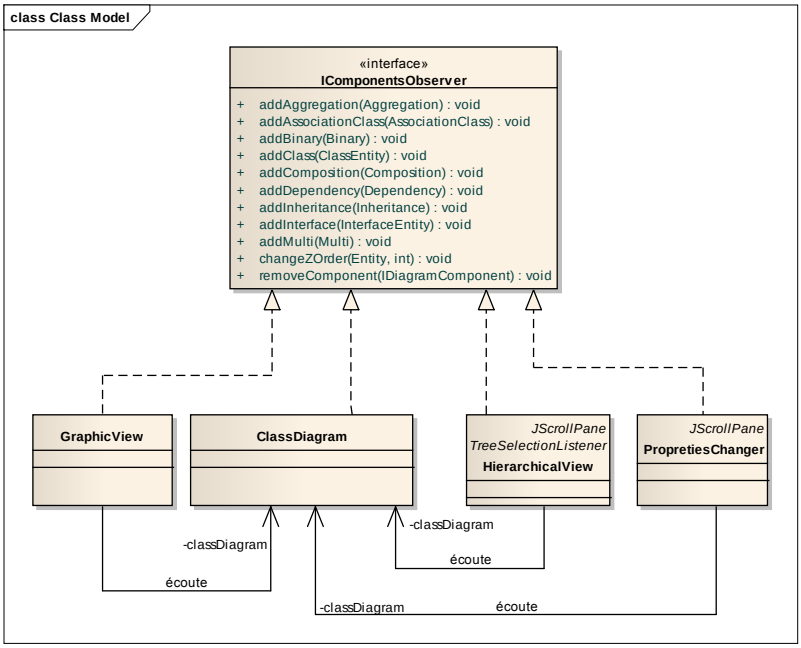


Illustration 13 Interface IComponentsObserver et classes l'implémentant

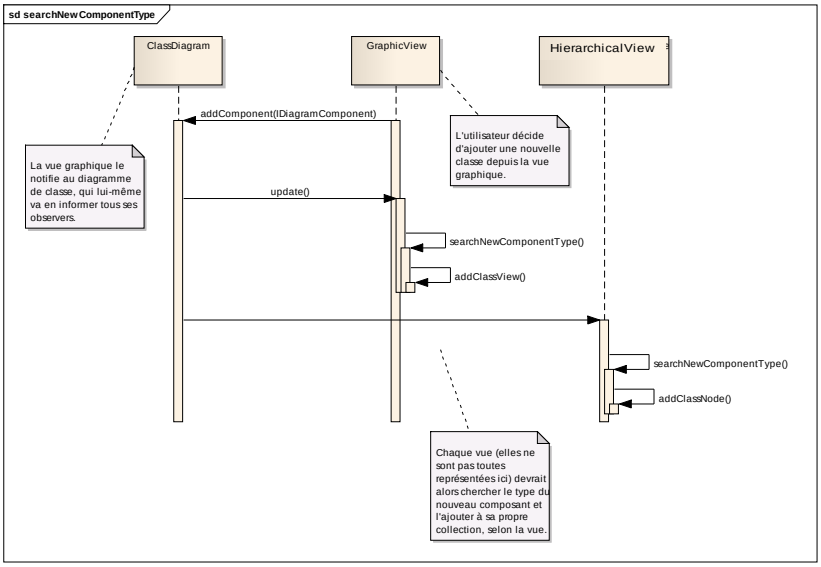


Illustration 14 Mécanismes de base lors de l'ajout d'une nouvelle classe

L'interface IComponentObserver permet de diviser la méthode *update* en plusieurs méthodes plus spécifiques. Ainsi, le diagramme de classes, au lieu de n'avoir que la méthode *update* à disposition pour notifier un changement, aura un jeu de méthodes plus spécifiques comme *addClass*, *addInterface*, *addInheritance*, *removeComponent* et même une méthode pour changer la position d'un élément dans le tableau (*changeZOrder*) (Illustration 15).

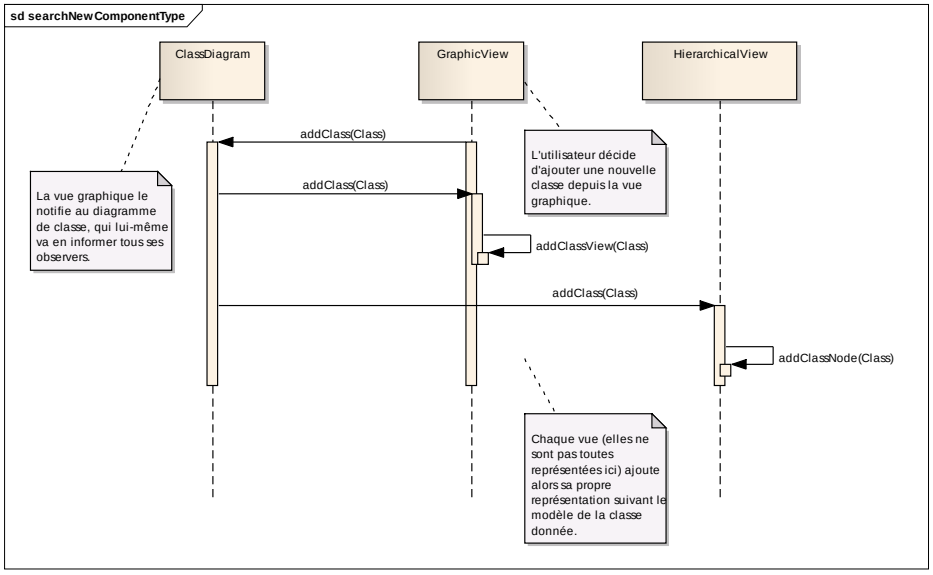


Illustration 15 Mécanisme implémenté lors de l'ajout d'une nouvelle classe

On voit que lors de l'ajout d'une nouvelle classe, le diagramme de classe va notifier toutes les vues et chaque vue va créer un objet représentant la nouvelle classe selon le type de représentation de la vue. Pour le GraphicView, il va créer un objet graphique représentant une classe UML. Pour la vue hiérarchique, il va créer un nouveau nœud pour son arbre, etc… . Chacune de ces nouvelles représentations va observer la classe de base comme expliqué au chapitre 5.1

4.1.1 Disposition des éléments (z-order)

La méthode changeZOrder() permet de définir l'ordre d'affichage des composants (qui n'est pas utile que pour la vue graphique). Effectivement, comme l'enregistrement du diagramme de classe se fait depuis le package classDiagram, il faut que l'ordre des composants de ce dernier soit identique à celui représenté sur la représentation graphique. Sinon, lors du chargement d'un fichier sauvegardé, l'ordre ne sera plus le même que celui représenté lors de l'enregistrement.

4.1.2 Suppression des éléments (removeComponent)

Une seule méthode suffit à la suppression des composants. Il suffit aux vues de chercher le composant fourni en paramètre (celui devant être supprimé) dans ses collections pour le supprimer.

4.2 Classe utilitaire (Utility)

La classe « Utility » est une classe utilitaire contenant des méthodes statiques utilisées dans différentes parties du projet, sans distinction particulière. La classe utilitaire se trouve dans le paquetage « graphic » car elle est principalement utilisée par les classes de ce paquetage. Cependant son contenu n'est en aucun cas dédié uniquement aux classes de ce paquetage.

C'est également dans cette classe que se trouve la classe interne « ImageSelection » qui va permettre d'enregistrer une image dans le presse-papier.

5 Meta-Schema de diagrammes de classes

Ce chapitre expose le meta-schema utilisé pour représenter les composants UML des diagrammes de classes. Exceptions faites que ce meta-schema ne couvre pas la totalité des possibilités fournies en UML, mais uniquement celles qui seront disponibles dans Slyum.

Paquetages du projet concernés :

- classDiagram.components -> Classes, Interfaces, Méthodes, Variables, etc.
- classDiagram.relationships -> Héritages, Associations, Compositions, agrégations, etc.

Le diagramme de classes complet est disponible dans les annexes.

5.1 IDiagramComponent

Cette interface est implémentée par tous les composants UML. Chaque élément UML possède des observateurs qui seront notifiés au moindre des changements de l'élément. La classe d'énumération UpdateMessage permet de cibler la modification apportée.

Lorsque l'on change le nom d'une classe depuis la représentation graphique du diagramme de classes, les changements doivent se répercuter sur toutes les autres vues. Dans la plupart des cas, le message de mise à jour (UpdateMessage) n'est pas fourni. Par exemple, pour le changement du nom de la classe, le nœud de l'arbre correspondant à la vue hiérarchique va se reconstruire en entier, cela prend peu de temps. Cependant, dans certains cas, comme l'ajout d'un attribut à une classe ne permet pas cette approche. En effet, si à chaque ajout d'attribut, il fallait régénérer en entier la représentation graphique de la classe, et que cette classe ait déjà un nombre de méthodes et d'attributs conséquent, cela signifierai qu'il faudrait redessiner l'intégralité de la classe. D'où l'importance du message de mise à jour dans certains cas.

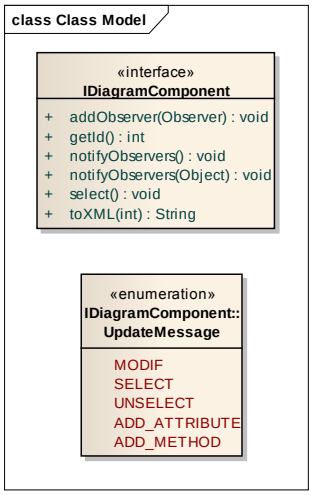


Illustration 16 Interface IDiagramComponent et sa classe d'énumération

Prenons l'exemple de la création d'une nouvelle classe, comme vu au chapitre 0, toutes les vues seront notifiées comme quoi une nouvelle classe vient d'être créée. Chaque vue va créer sa propre représentation de la classe avant de l'y abonner (Illustration 17). Lorsque la classe change de nom, par exemple, toutes les représentations associées à cette classe seront averties et vont répercuter ces changements.

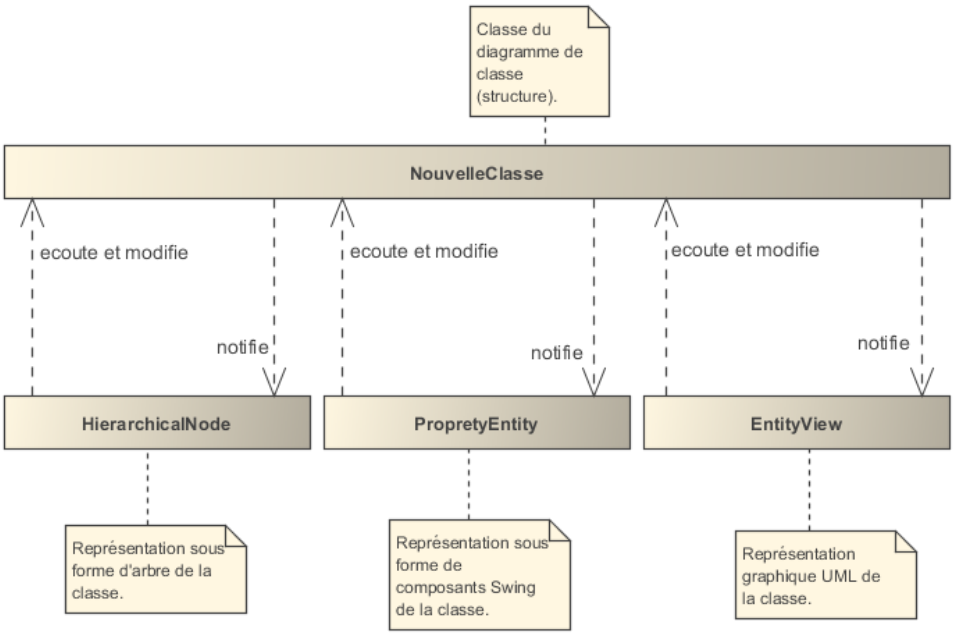


Illustration 17 Mécanisme de notification des éléments UML

5.2 Diagramme de classes (package classDiagram)

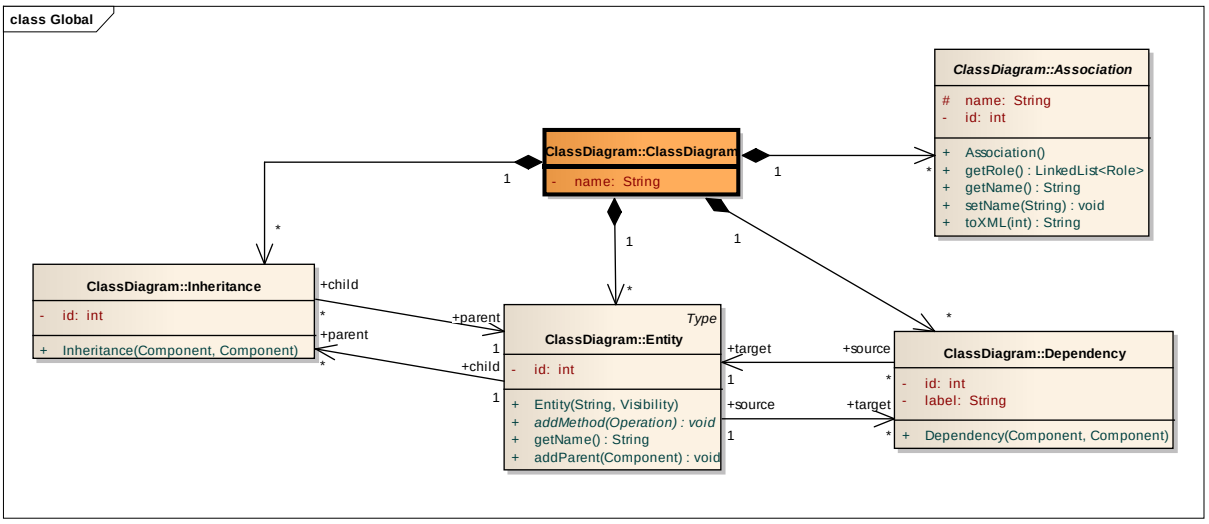


Illustration 18 Diagramme de classes du package classDiagram

Le diagramme de classes ci-dessus illustre les principales classes intervenant dans la structure objets du meta-schema. Seule la classe *ClassDiagram* fait partie de ce package.

ClassDiagram

ClassDiagram est la classe principale du meta-schema, elle regroupe toutes les entités (classes et interfaces), relations, dépendances et associations de la structure du système. Elle permet entre-autre d'ajouter, supprimer ou modifier les entités et les relations entre-elles.

5.3 Composants (package components)

Ce package contient toutes les classes nécessaires à la création des entités d'une structure orientée objet. Il permet de créer des :

- attributs,
- classes,
- interfaces,
- méthodes,
- types et types primitifs,
- variables,
- visibilités.

Diagramme de classes

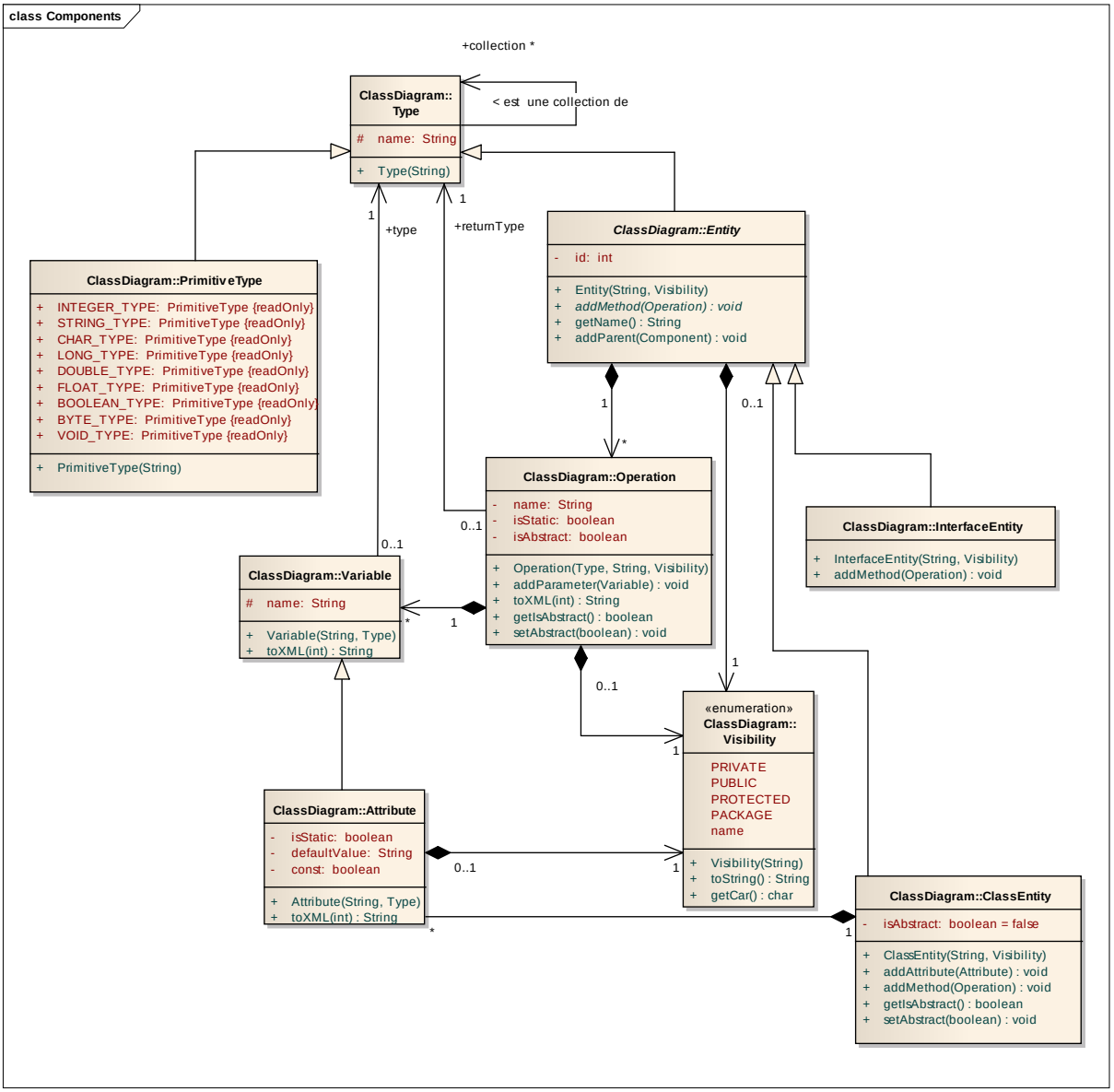


Illustration 19 Diagramme de classes du package classDiagram.components

5.3.1 Type

Cette classe, tout en haut de la hiérarchie, représente un type de données. Un type à une association sur lui-même afin de permettre la création d'une collection de ce même type (qui représente, en programmation, un tableau). Un type peut être soit primitif (PrimitiveType) soit une classe (Entity).

5.3.2 PrimitiveType

Les types primitifs sont les types de bases par défaut dans Slyum et dans la plupart des langages de programmations.

Integer	**String**	**Char**	**Long**	**Double**
Float	**Boolean**	**Byte**	**Void**	

5.3.3 Visibility

Cette classe d'énumération permet de définir la visibilité d'une classe, interface, opération ou attribut. Elle possède les valeurs suivantes : « Private », « Public », « Protected », « Package ».

5.3.4 Entity

La classe abstraite *entity* regroupe les deux éléments principaux des diagrammes de classes ; les classes et les interfaces. Le mot entité a été choisi pour représenter, sans distinction, soit une classe, soit une interface ou encore une classe d'association.

Une entité possède une liste d'opérations (*Operation*) ainsi qu'une visibilité (*Visibility*). Un attribut définit son nom.

5.3.5 ClassEntity

Enfant d'*Entity*, cette classe représente … une classe. Son existence n'a pas vraiment lieu d'être puisqu'elle ne modifie aucun comportement de son parent. Cette classe existe pour rendre la structure plus compréhensible.

5.3.6 AssociationClass

Une classe d'association est une classe normale ayant une référence sur une association binaire. L'association binaire peut être créée depuis la classe d'association, ou déjà exister et simplement lui ajouter une classe d'association par-dessus.

5.3.7 InterfaceEntity

Deuxième enfant d'*Entity*, cette classe représente une interface. Elle est en tout point semblable à son parent. Elle redéfinit les méthodes permettant d'ajouter des attributs et des méthodes en s'assurant qu'ils soient corrects. Dans le cas contraire, elle modifie la nouvelle méthode / le nouvel attribut pour qu'il puisse être ajouté correctement. Concrètement, elle va rendre abstraite toutes les méthodes qu'on lui ajoute, et rendre statique tous les nouveaux attributs.

5.3.8 Operation

Une opération a un nom (*String*) et un type (*Type*) de retour. Elle peut être abstraite ou non. Elle possède une liste de variables (*Variable*) représentant ses paramètres ainsi qu'une visibilité (*Visibility*).

5.3.9 Variable

Représente une variable avec un nom (*String*), un type (*Type*) et une visibilité (Visibility).

5.3.10 Attribute

Enfant de la classe *Variable*, *Attribute* représente un attribut d'une classe. Elle possède, en plus du nom (*String*) et du type (*Type*), une visibilité (*Visibility*), une valeur par défaut et la possibilité d'être statique ou constante.

5.4 Relations (package relationships)

Ce package contient toutes les classes nécessaires à la création des relations. Il permet de créer des :

- agrégations,
- associations binaires,
- compositions,
- dépendances,
- relations d'héritages,
- multi-associations,
- multiplicités,
- rôles.

Diagramme de classes

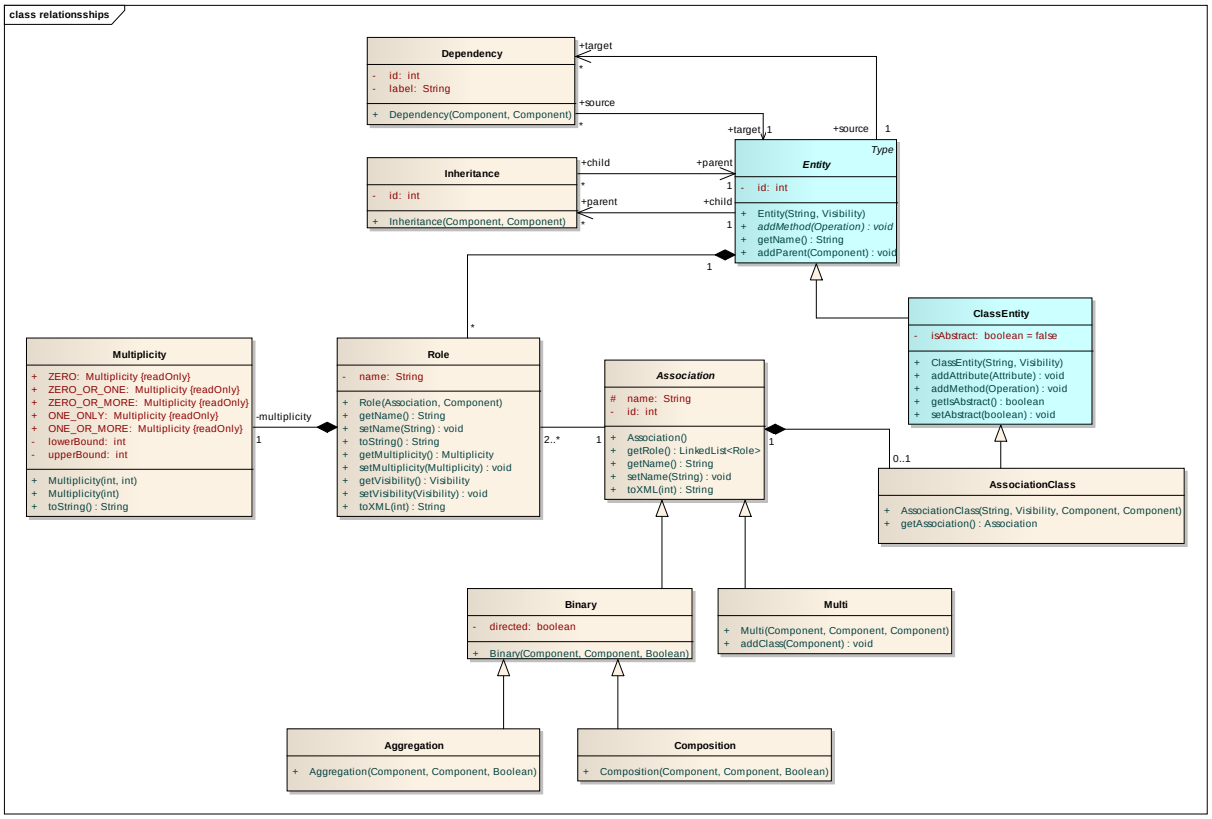


Illustration 20 Diagramme de classes du package relationships

Remarque : Pour plus de lisibilité, certaines classes du package *components* apparaissent également dans ce diagramme, elles sont représentées en bleu.

Multiplicity

Cette classe permet la création des multiplicités utilisées dans les associations. Elle possède une valeur minimale et une valeur maximale. La valeur *Integer.MAX_VALUE* est utilisée pour représenter une borne supérieure infinie. Les multiplicités courantes existent par défaut dans Slyum.

Zéro	0
Zéro ou une	0..1
Zéro ou plusieurs	0 .. * ou *
Une	1
Au moins une	1 .. *

Role

Un rôle possède une multiplicité (*Multiplicity*) et se rapporte à une entité (*Entity*). Une association possède pour chacune de ses extrémités un rôle. Les rôles font les liens entre l'association et les entités participant à l'association. Il y a un rôle par entité participante et ce rôle possède un nom, une visibilité (*Visibility*) et une multiplicité.

Association

Cette classe abstraite représente une association (au sens UML). Elle possède deux rôles au minimum qui font le lien avec les entités qu'elle relie. Elle peut posséder plus de deux rôles, dans ce cas elle devient une association multiple (*Multi*), au contraire, si elle n'en possède que deux, c'est une association binaire (*Binary*).

Binary, Composition, Aggregation, Multi

Ces enfants d'*Association* permettent de préciser de quel type d'association il s'agit. La classe *Binary* et ses enfants possèdent un attribut *boolean* indiquant si l'association est dirigée ou non. La classe multi possède une méthode permettant d'ajouter de nouveaux rôles (et donc indirectement de nouvelles entités) participant à l'association.

AssociationClass

Cette classe permet d'associer une classe à une association. Elle étend *ClassEntity* en y ajoutant une référence sur l'association à qui elle se rapporte. De cette manière nous pouvons ajouter une classe d'association avec toutes les possibilités d'une classe (attributs, opérations, etc…) à n'importe quel type d'association (binaire, composition, agrégation, multi).

Dependency et Inheritance

Ces relations sont plus simples que les associations, elles possèdent uniquement une entité source (ou enfant) et une entité cible (ou parente) avec, éventuellement, un label (uniquement pour les dépendances).

6 Représentation graphique

La représentation graphique du diagramme de classes est faite à l'aide de la structure présentée dans ce chapitre. Les composants graphiques n'étendent pas de classes graphiques Swing, ils ont été entièrement refait afin de permettre une représentation fidèle aux diagrammes de classes. Le seul composant Swing existant (pour la représentation du diagramme de classe) est un *JPanel*. Tous les autres composants sont gérés manuellement (événement, dessin, déplacement, agrandissement, etc.).

La structure du projet pour cette partie est incluse dans le paquetage *graphic* et ses sous-packages.

- graphic -> Éléments principaux utilisés dans les sous-packages.
- graphic.entity -> En rapport avec les classes et interfaces.
- graphic.factory -> Création de nouveaux éléments graphiques.
- graphic.relations -> En rapport avec les relations.
- graphic.textbox -> Affichage de texte pouvant être modifié (attributs, rôles, commentaires…)

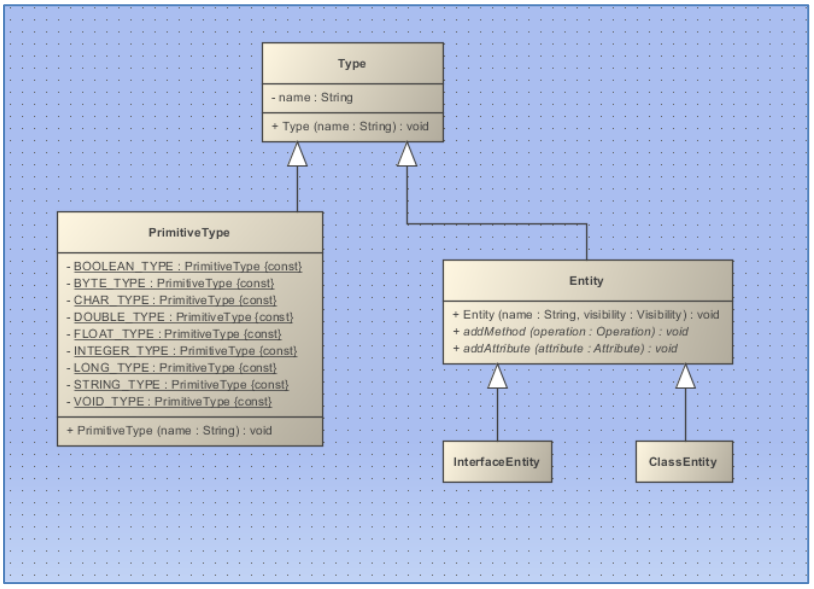


Illustration 21 Exemple de représentation d'un diagramme de classe avec Slyum

6.1 Structure de la représentation graphique

Avant de détailler le contenu des paquetages utilisés dans la représentation graphique des diagrammes de classes, voyons la structure globale et les principaux composants entrant en scène.

Le schéma de la page suivante (Illustration 22) est une représentation de l'ensemble des classes de la bibliothèque graphique regroupées par catégories. Chaque catégorie (différenciées par couleur) représente un ensemble d'objets de la même famille de la bibliothèque graphique. En rouge, la classe mère de tous les composants graphiques de Slyum, se nomme GraphicComponent. La classe GraphicView, en rose, est la classe contenant le *JPanel* et celle qui va gérer tous les autres GraphicComponent. En violet, LineView et ses enfants servent à la représentation des relations. Les TextBox, de couleur verte, sont des zones affichant un texte pouvant être édité. Les SquareGrip sont de petits carrés gris déplaçables par l'utilisateur pour modifier ou redimensionner des composants. En jaune se trouvent les MovableComponent qui regroupent les classes, interfaces, et autres éléments UML autres que les relations. Pour finir les CreateComponent permettent de créer de nouveaux composants graphiques.

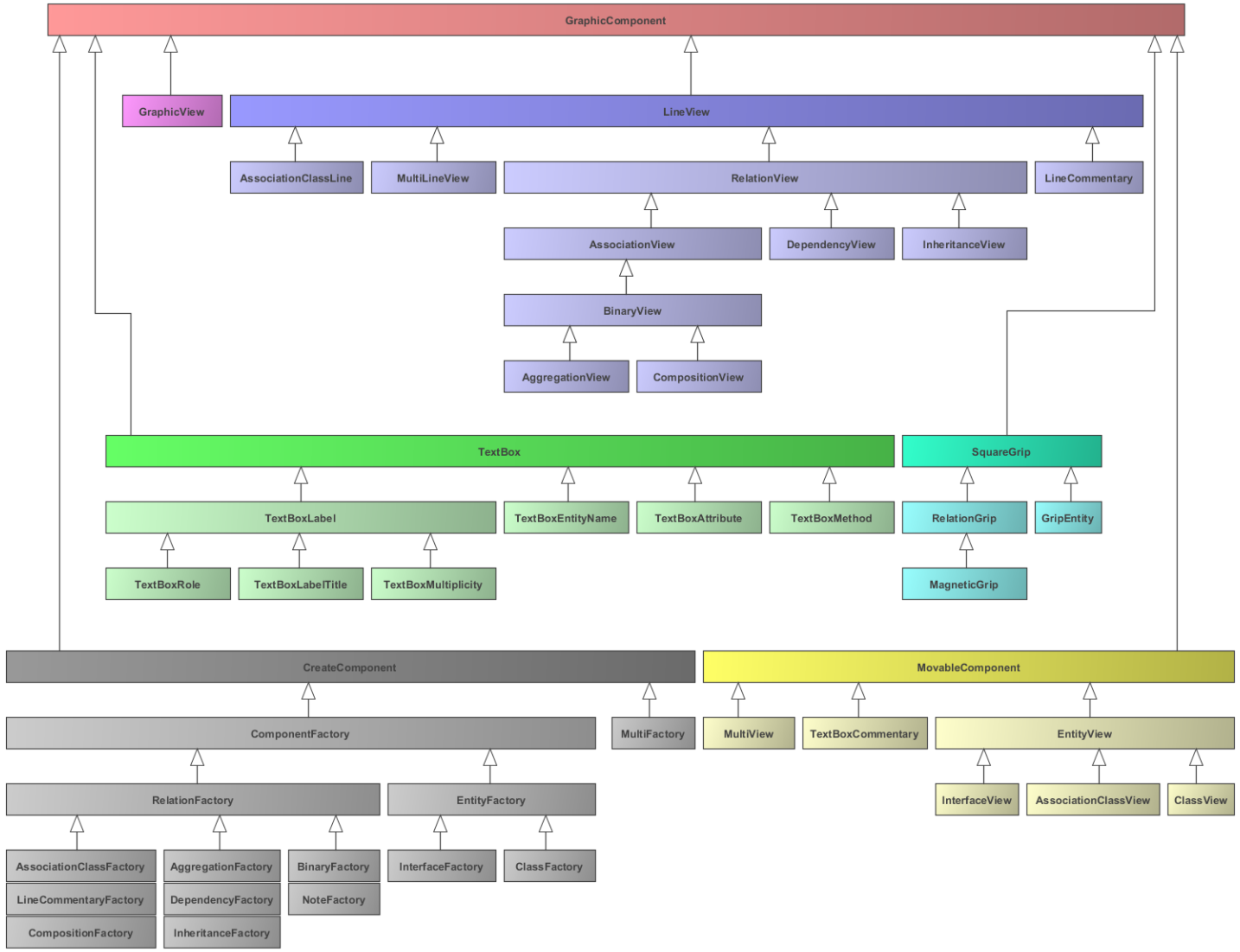


Illustration 22 Structure complète de la bibliothèque graphique

6.2 Composant graphique (GraphicComponent)

N'importe quel composant graphique de Slyum doit étendre la classe *GraphicComponent*. Étendre cette classe permet de gérer les événements souris, sa représentation graphique ou encore sa position sur la zone graphique (GraphicView).

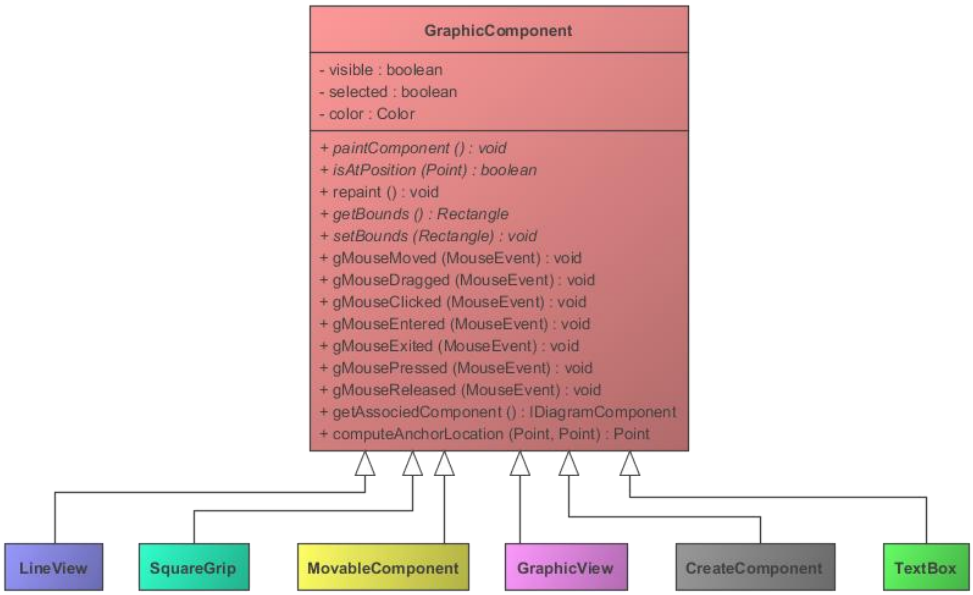


Illustration 23 Classe GraphicComponent et ses principales méthodes

Pour créer un nouvel élément graphique, il faut étendre la classe *GraphicComponent* et redéfinir ses méthodes abstraites.

```
public abstract void paintComponent(Graphics2D g2);
```

C'est la méthode qui sera appelée lorsque le dessin de la scène est effectué. C'est dans cette méthode que la représentation graphique de l'objet se fait, en utilisant l'objet *Graphics2D*.

```
public abstract boolean isAtPosition(Point position);
```

Cette méthode doit retourner si le *Point* passé en paramètre se trouve dans l'objet graphique ou non. Elle est utilisée par le *GraphicView* pour savoir sur quel composant la souris se trouve.

```
public abstract void repaint();
```

Demande de redessiner la scène à l'endroit spécifié (en utilisant, en général, les limites (*bounds*) du composant).

```
public abstract Rectangle getBounds();
public abstract void setBounds(Rectangle bounds);
```

Permet de connaître et définir les limites du composant graphique. Cependant, certains composants, comment les relations, n'utilisent pas de rectangle comme limites (on ne peut pas représenter une relation ayant plusieurs lignes avec un seul rectangle). Ce qui signifie que certains composants graphiques ne réagissent pas à la méthode setBounds().

```
public void getAssociedComponent();
```

Par default cette méthode return **null**. Elle est redéfinie par tous les composants graphiques ayant un composant de la structure UML associée (comme une classe, une interface, un attribut ou encore une association) et retourne cet élément UML. Cette méthode permet facilement de savoir si un composant est associé à un élément structurel UML ou non en regarde sa valeur de retour. Cependant, une amélioration de la structure objet serait de diviser les composants graphiques ayant un élément UML associé et ceux n'en ayant pas.

```
public void computeAnchorLocation(Point, Point);
```

Cette méthode est utilisée par les MagneticGrip pour calculer leur position. Se rapporter au chapitre sur les MagneticGrip (dasdasd) pour plus d'informations.

6.2.1 Gestion des événements

Les méthodes préfixées de la lettre « g » sont les équivalents des événements Swing et ne font rien tant qu'ils n'ont pas été redéfinis. Par exemple, pour affecter une action spéciale lorsque l'utilisateur clique sur un composant, il faut redéfinir sa méthode *gMouseClicked*.

Créer un GraphicComponent ne permet pas de l'afficher et de le faire fonctionner car il ne se suffit pas à lui-même. Il doit être géré par une classe possédant un élément Swing qui fera le lien entre les composants graphiques Slyum et les composants graphiques Swing. Cette classe s'appelle GraphicView et possède un JPanel pour faire cette liaison.

Lorsque l'on crée un nouveau GraphicComponent, il est obligatoire de lui donner une référence sur un GraphicView. Cette obligation vient du fait que le GraphicComponent se gère lui-même. Il a besoin d'une référence sur son parent[1] (le GraphicView) pour demander à la scène de se redessiner, pour connaître les autres éléments graphiques ou encore pour se supprimer.

[1] Le terme parent est utilisé ici au sens événementiel et non au sens POO.

6.3 Vue graphique (GraphicView)

Voici la représentation des principales méthodes et attributs de la classe GraphicView (les méthodes redéfinies ne sont pas affichées).

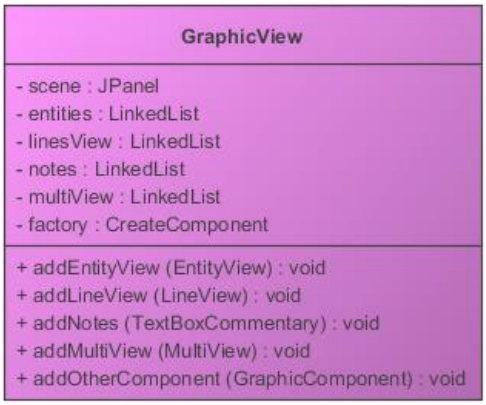


Illustration 24 Classe GraphicView

GraphicView est une vue, ce qui signifie qu'elle implémente IComponentObserver et qu'elle sera notifiée à chaque changement du diagramme de classe. Lors de l'ajout d'une nouvelle classe par exemple, elle va créer un nouvel objet de type ClassView associé à cette nouvelle classe.

GraphicView est la classe motrice de la représentation graphique grâce à son objet de type *JPanel* (Swing). Comme le *GraphicView* est aussi un élément graphique (il va afficher une grille, une couleur de fond et permettre de faire un rectangle de sélection), elle étend *GraphicComponent*. *GraphicView* et tous ses contenants se dessinent sur le *JPanel*. La taille du *JPanel* est la même que celle du *GraphicView*. Java ne permettant pas l'héritage multiple, *GraphicView* étend uniquement *GraphicComponent*, cependant, on peut considérer, visuellement, ces deux objets (*JPanel* et *GraphicView*) comme n'en faisant qu'un.

6.3.1 Gestion des collections des composants graphiques

GraphicView propose des méthodes pour ajouter et supprimer des composants (de type *GraphicComponent*) dans ses collections. Lorsque l'on ajoute un nouveau composant, celui-ci sera automatiquement dessiné et ses événements souris seront gérés. Pour ajouter un nouveau GraphicComponent, la classe dispose de plusieurs méthodes (voir Illustration 24). Les entités, les relations, les notes ou encore les multi-associations ont chacun leur propre méthode pour s'ajouter au GraphicView. Enfin si vous désirez ajouter un composant graphique n'appartenant pas à ces catégories, la méthode *addOtherComponent*() est faite pour.

La raison de ces différentes méthodes est de garder les types de composants séparés. Si tous les composants étaient réunis au sein d'une même LinkedList (comme ce fut le cas au début), le contrôle sur ces différents composants serait judicieux. Maintenant, le GraphicView possède une collection pour chacun de principaux composants graphiques. À savoir :

- EntityView (vue des classes, interfaces et classe d'association)
- LineView (relations)
- TextBoxCommentary (notes)
- MultiView (vue des multi-associations)
- Others components (tout autre GraphicComponent)

6.3.1.1 *Ordre d'affichage des composants*

Ce nouveau système remplace l'ancien système de calques mis en place.

Le nouveau système pour gérer l'ordre d'affichage des composants est tout simplement définit par le type du composant graphique. Voici l'ordre d'affichage des composants (les composants du haut de la liste seront dessinés par-dessus les composants au-dessous.

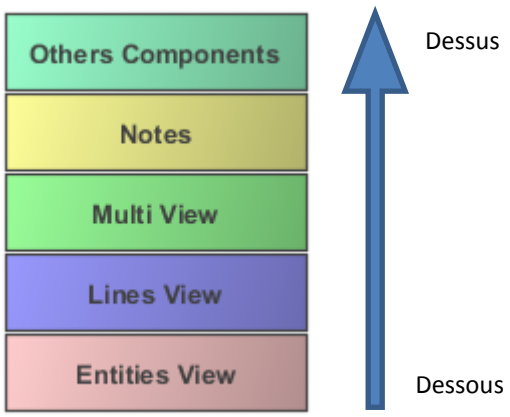


Illustration 25 Ordre d'affichage des composants

6.3.2 Gestion des événements

GraphicView écoute les événements souris et clavier du *JPanel* pour ensuite les envoyer sur l'élément de type *GraphicComponent* adéquat, qui peut très bien être lui-même (le *GraphicView*). Elle utilise la méthode *isAtPosition()* de tous les composants qu'elle contient pour savoir sur quel composant rediriger l'événement. La méthode getComponentAtPosition(Point) permet de trouver le composant se trouvant sur le point fourni en paramètre. Si plusieurs composants se trouvent sur le point, c'est l'ordre d'affichage qui définit quel composant sera retourné.

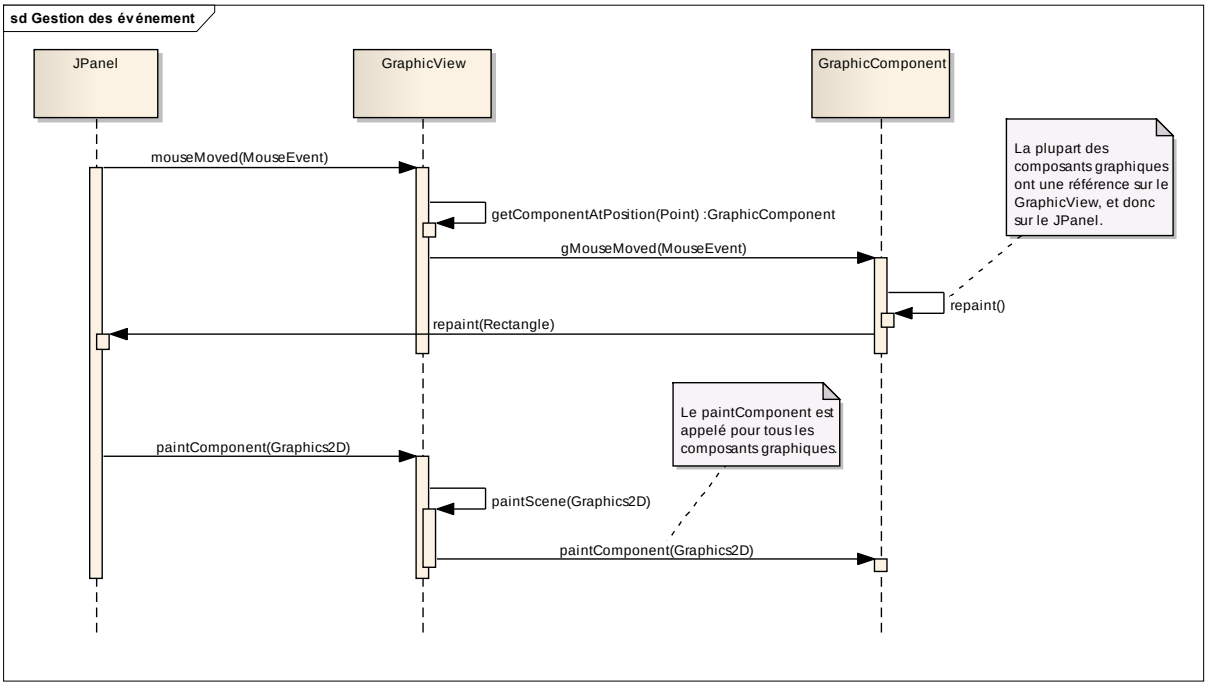


Illustration 26 Exemple de fonctionnement de la gestion des événements

gMouseEnter et gMouseLeave

Les événements *mouseEnter* et *mouseLeave* correspondent, respectivement, à l'événement appelé lorsque la souris entre, ou sort, d'un composant. Ces deux événements ne peuvent pas être redirigés car ils sont appelé uniquement lorsque la souris entre (ou sort) du *JPanel*. Pour résoudre ce problème, ils sont calculés dynamiquement lorsque la souris bouge sur le *JPanel* grâce à l'opération *computeComponentEventEnter()*. Cette méthode enregistre le dernier composant sur lequel la souris se trouvait et le compare avec le composant actuel survolé par la souris. S'ils sont différents, l'événement *gMouseLeave* va être appelé pour le composant sauvegardé et *gMouseEnter* sur le composant actuel.

6.3.3 Calques (layer) – Ce système n'existe plus

Ce système de calques a été remplacé par le système du chapitre 6.3.1.1. Je me suis rendu compte qu'au final, toujours le même type de composant se trouvait sur un niveau donné. Si bien que je me permettais de sélectionner un ensemble de composants graphiques en misant sur le fait qu'ils étaient sur une couche donnée. Mais comme aucun contrôle n'était fait (rien n'interdisait de mettre une entité dans une couche supérieure), il se pouvait qu'en changeant ma stratégie dans le futur le programme ne détecte plus certains composants.

Le *GraphicView* dispose d'un système de calque pour contrôler l'ordre d'affichage des composants. L'ordre d'affichage des composants influe sur leur superposition ; quel composant sera au-dessus d'un autre.

Lors de l'ajout d'un nouveau composant, un paramètre permet de spécifier sur quel niveau de la couche le composant sera ajouté. Les couches supérieures sont dessinées en dernier, et donc seront au-dessus des composants des couches inférieures.

Ce système est défini au niveau du programme (l'utilisateur ne peut pas l'influencer) et est structuré ainsi :

1. Entité et leur contenu (classes, interfaces, attributs, méthodes, etc.).
2. Relations (associations, héritages, etc.).
3. Grips (les grips sont expliqués au chapitre suivant).

Ce système est mis en place pour permettre aux relations de toujours se trouver au-dessus des entités, et les grips au-dessus de tous les autres composants. De cette manière, l'utilisation du programme est plus agréable pour l'utilisateur ; des composants difficilement sélectionnable (comme les relations ou les grips) se trouveront toujours au-dessus des composants faciles à sélectionner (comme les classes ou les interfaces).

Il ne faut pas confondre ce système avec le z-order (qui n'est pas encore implémenté). Le z-order définit, tout comme les calques, l'ordre d'affichage des composants, mais au niveau de chaque composant et non d'une collection de composants de la même famille. C'est un système qui peut être paramétré par l'utilisateur.

6.4 Carrés gris (graphic.SquareGrip)

La famille des carrés gris (…) représente de petits carrés que l'utilisateur peut déplacer avec la souris. Il y en a pour redimensionner certains composants ou encore pour modifier la trajectoire d'une relation.

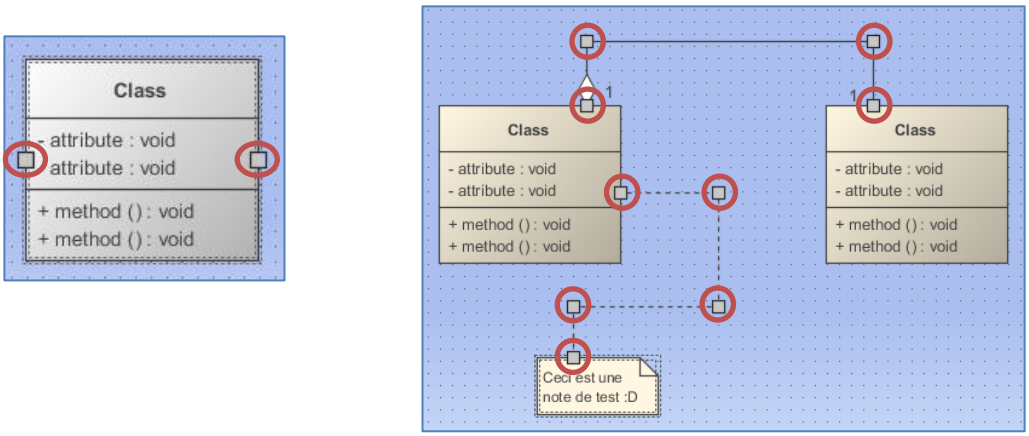


Illustration 27 Exemple de carrés gris (entourés en rouge)

6.4.1 Structure des carrés gris

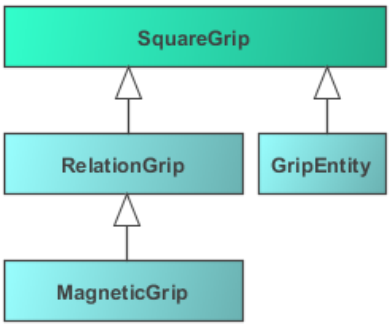


Illustration 28 Structure des classes de la famille des carrés gris

6.4.2 Classe de base (SquareGrip)

La classe abstraite *SquareGrip* définit les actions de bases de ces composants. Elle n'a par contre absolument aucune utilité en tant que tel (d'où le fait qu'elle soit abstraite).

La première chose accomplie par cette classe est sa représentation graphique. Tous les carrés gris ont la même apparence définie dans la classe de base. Qui est simplement un carré gris avec une bordure noire.

Elle va également redéfinir deux événements de souris : gMouseEntered et gMouseExited. Par défaut, un carré gris n'est pas visible. Il sera uniquement visible lorsque l'utilisateur passera la souris dessus, et redeviendra invisible lorsque la souris le quitte (dans son comportement de base tout du moins).

6.4.3 Les carrés gris des entités (GripEntity)

Cette classe est également abstraite, malgré le fait qu'elle se trouve tout en bas de la hiérarchie. Elle utilise les utilise les mécanismes de déplacement et de redimensionnement mis en place dans les composants graphiques graphiques (0

Fonctionnement des composants déplaçables). Elle est abstraite car un composant redimensionnable possède deux GripEntity. Le premier sert pour le redimensionnement depuis la gauche, et le second depuis la droite. Cependant la méthode de redimensionnement d'un composant graphique est laissée à ce dernier. C'est pour cela que deux classes anonymes sont requises ; la première appelant la méthode permettant de redimensionner un composant depuis la gauche, et la seconde depuis le droite. Le code étant relativement court (il se résume à la redéfinition d'une méthode pour en appeler une autre) des fichiers de classes n'ont pas été créés pour l'occasion.

Concrètement, chaque composant graphique possède deux méthodes pour le redimensionnement ; l'une à gauche et l'autre à droite (seuls les *MovableComponent* redéfinissent cette méthode, les autres composants n'en font rien). Le but est de créer une classe anonyme redirigeant l'événement gMouseDragged d'un GripEntity sur la méthode de redimensionnement appropriée pour le faire fonctionner.

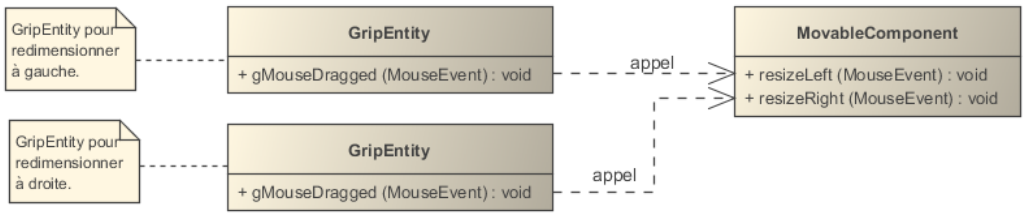


Illustration 29 Fonctionnement des GripEntity

Comme les GripEntity ne sont pas associé à un composant (c'est les composants qui possèdent des GripEntity), il revient au composant les possédant de les positionner correctement.

Cette méthode oblige chaque GraphicComponent à posséder des méthodes que peu utilisent. Une amélioration serait de créer des carrés gris en rapport avec certains composants (comme les MovableComponent). Les GripEntity en seraient moins générique mais la structure serait plus propre (c'est le grip qui saurait comment redimensionner la classe qui lui est attribuée, et pas la classe elle-même).

6.4.4 RelationGrip

Les RelationGrips sont les grip associés aux relations. Tous les RelationGrip possèdent une référence sur une relation et l'informe lorsque le grip est déplacé (via la méthode *gripMoved()*). Une relation possède une collection de RelationGrips qui vont définir sa trajectoire. En effet, pour se dessiner, une relation va tirer une ligne entre chaque RelationGrip qu'elle possède.

Pour convenir aux exigences des relations, les RelationsGrips modifient leurs limites (*bounds*) pour en faire qu'un point (*anchor*). Ce point sera simplement le milieu de leurs limites.

$$\text{anchor} = (bounds.x + \frac{bounds.width}{2}, bounds.y + \frac{bounds.height}{2})$$

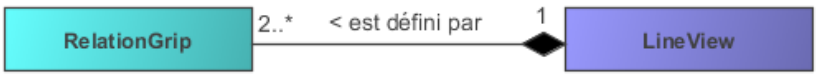


Illustration 30 Relation entre RelationGrip et LineView

6.4.5 MagneticGrip

Les grips magnétiques, hérité de RelationGrip, sont les grips des extrémités des relations (Illustration 31).

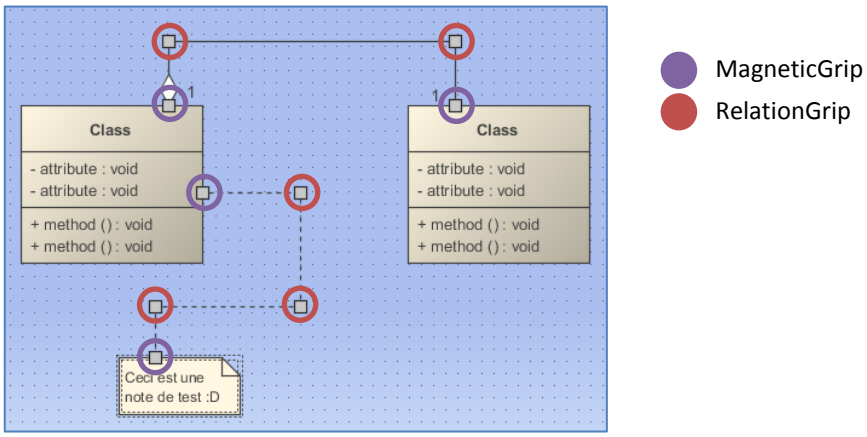


Illustration 31 Comparaison entre RelationGrip et MagneticGrip

Ce que les grips magnétiques ont en plus des RelationGrip est un GraphicComponent sur lequel ils sont magnétisés. Comme un grip magnétique peut se placer sur n'importe quel composant graphique, il n'était pas nécessaire de faire un type de grip magnétique pour chaque composant graphique, mais plutôt d'adapter les composants graphiques à recevoir des grips magnétiques. C'est pour cette raison qu'a été ajoutée la méthode *computeAnchorLocation()* à la classe GraphicComponent. Cette méthode prend en paramètre deux points ; first et next, permettant le calcul de la position du grip. Lorsqu'un grip veut se magnétiser sur le composant qui lui est associé, il appel ladite méthode de ce composant pour connaître la position dans l'espace où il doit se placer.

Le paramètre first correspond à la position actuelle du grip magnétique et next à la position du prochain RelationGrip de la relation (explications au chapitre suivant).

Une dernière information importante sur les grips magnétiques est leur mode, défini par une variable boolean, qui va indiquer si un grip est magnétisé ou non. Lorsque l'on change le point d'ancrage (*setAnchor(Point)*) d'un grip magnétique, celui-ci va le modifier si il est magnétisé, ou le laisser tel quel sinon. Un exemple typique est quand l'utilisateur déplace le grip avec la souris, il n'est plus magnétisé. Dès que l'utilisateur relâche le bouton de la souris, il redevient magnétique.

6.4.6 Méthodes pour calculer la position des grips magnétiques

Comme expliqué au chapitre précédent, la position des grips magnétiques se calcule en fonction du composant graphique qui lui est associé. Les prochains chapitres exposent les différentes méthodes de calcules qui existent pour le placement de ces grips.

6.4.6.1 *Par défaut*

Par défaut, la méthode *computeAnchorLocation()* des GraphicComponent retourne le milieu des limites du composant. C'est-à-dire que le grip va être magnétisé et se trouvera toujours au milieu du composant, selon la formule :

$$(bounds.x + \frac{bounds.width}{2}, bounds.y + \frac{bounds.height}{2})$$

Les deux paramètres *first* et *next* ne sont pas utilisés par défaut.

6.4.6.2 *Des MovableComponent (forme rectangulaire)*

Les MovableComponent, plus généralement les classes, interfaces, classes d'associations ou encore les notes (qui sont tous de forme rectangulaire), calcule la projection du point *first* vers le point *next* sur l'un des côté du rectangle.

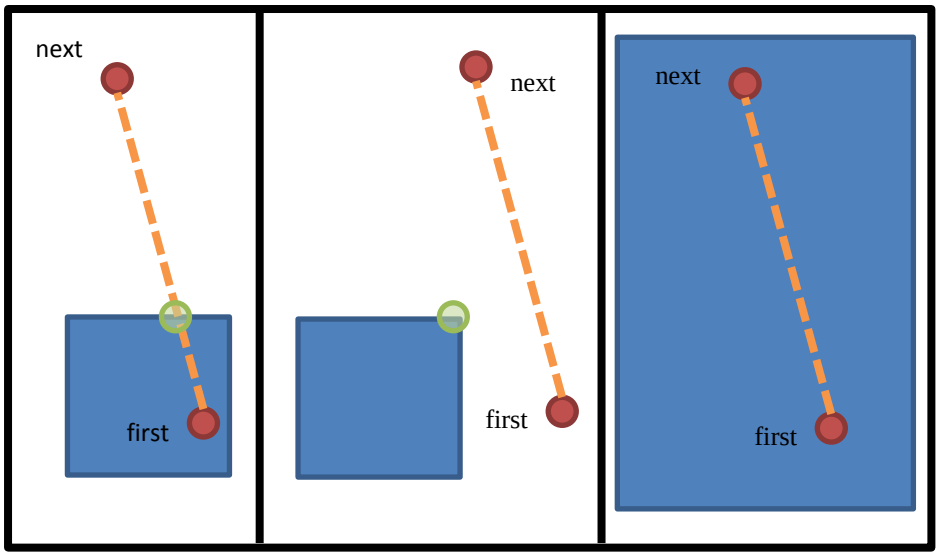


Illustration 32 Calcule de l'ancrage pour un rectangle

Dans le premier cas (à gauche), le point retourné par la méthode *getAnchorLocation()* sera le point représenté en vert. Dans le cas du milieu, ce sera le point le plus proche trouvé. Dans le dernier cas, le programme détecte que l'utilisateur veut faire une association récursive et va agir en tant que tel. Il le détecte tout simplement car la méthode retournera la valeur *null*.

6.4.6.3 *Des relations (collections de lignes)*

Dans le cas d'une relation, elle calcule le point sur ses segments le plus proche du point *first*.

6.4.6.4 *De la vue graphique (GraphicView)*

La vue graphique return la valeur *next*. Ce qui signifie que les deux le grip magnétiques seront placés au même endroit que leur voisin. Cependant, dans Slyum, une vérification est faite lorsqu'un grip magnétique essaye de se magnétiser avec la vue graphique engendrant la suppression pure et simple de la relation à laquelle il appartient.

6.4.6.5 *D'une multi-association (losange)*

Une multi-association, en forme de losange, permet aux grips magnétique de se positionner à l'un de ses quatre coins. En utilisant *next* et *first*, la multi-association va chercher le coin le plus adapté à la situation de la manière suivante.

1. Pour chaque coin
    a. Cherche la distance entre *first* et le coin.
    b. Cherche la distance entre *next* et le coin.
    c. Addition les deux distances trouvées aux points a et b.
2. Recherche la plus petite distance calculée au point c.
3. Retourne le coin ayant la plus petite distance trouvée au point 2.

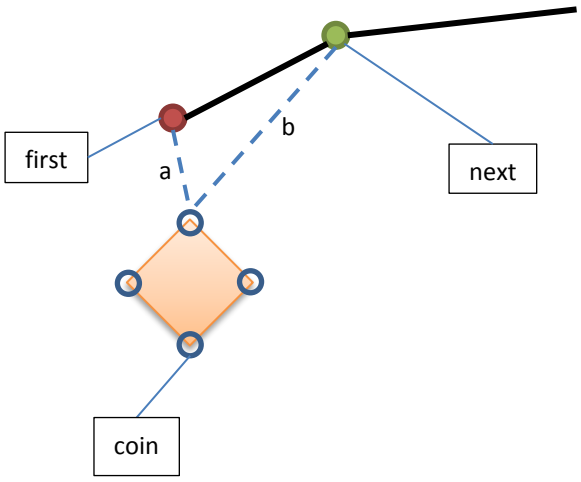


Illustration 33 Calcule du coin pour une multi-association

6.4.7 Changement de composant d'une relation

Lorsque l'utilisateur place un grip magnétique sur un autre composant que celui sur lequel il est associé, le grip va faire une demande pour savoir si le changement de composant est légal ou non.

Tous les composants de la famille LineView (les relations) possèdent une méthode :

```
relationChanged(GraphicComponent, GraphicComponent) : boolean
```

Chaque fois que l'utilisateur déplace un grip magnétique, celui-ci va regarder sur quel composant il est placé. Si ce n'est pas le composant auquel il est associé actuellement, il va demander à la relation à laquelle il appartient si le nouveau composant est compatible avec la relation en appelant la méthode ci-dessus. Le premier paramètre est le composant actuel associé au MagneticGrip. Le second paramètre est le nouveau composant que l'utilisateur essaye d'associer au MagneticGrip. Si la méthode retourne vraie, le grip magnétique va changer le composant auquel il est magnétisé par le nouveau, sinon il va se repositionner en fonction de l'ancien composant.

Remarque : Il est du devoir de la relation d'agir en conséquence lorsqu'elle retourne vraie. Si, par exemple, l'utilisateur change les classes participant à une relation d'héritage, la relation doit, avant de retourner vraie, modifier la structure interne du diagramme de classe en conséquence pour qu'il reste cohérent avec sa représentation graphique.

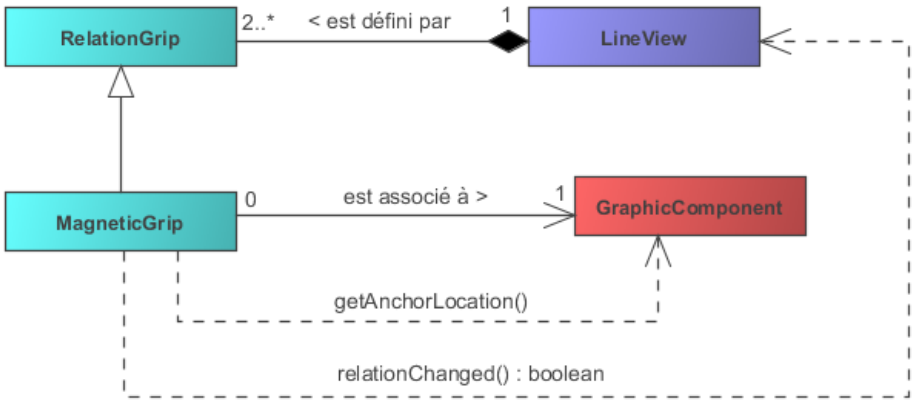


Illustration 34 Composants associés aux grips magnétiques

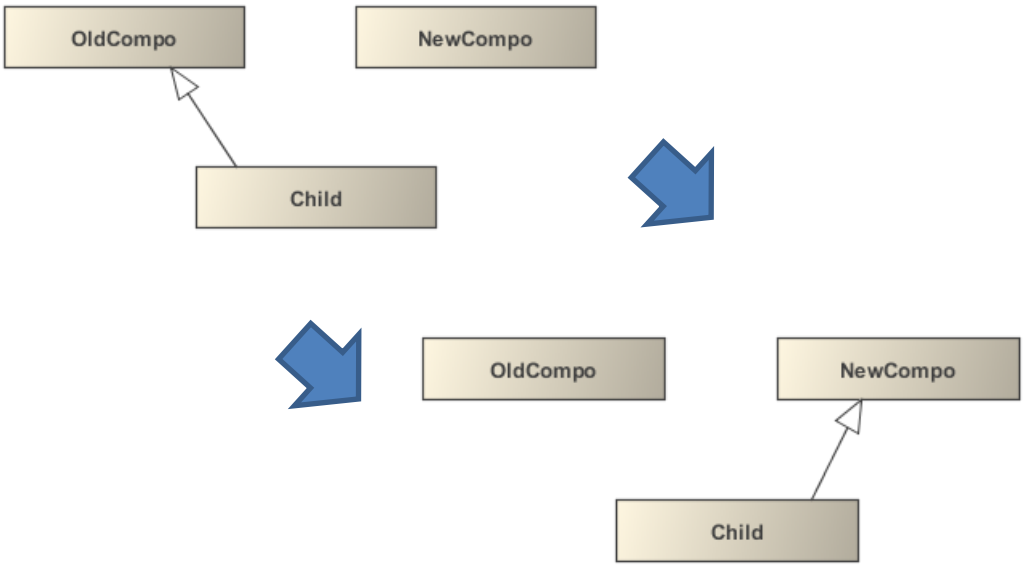


Illustration 35 Représentation du changement du composant associé à un grip magnétique

Dans l'exemple ci-dessus, le grip magnétique placé à l'extrémité de la flèche va appeler la méthode de la relation d'héritage avec comme paramètre :

```
relationChanged(oldCompo, newCompo);
```

La relation va se charger de modifier la structure du diagramme de classe (package classDiagram) en changeant le parent de la classe Child par NewCompo. Puis retournera vraie pour permettre au grip magnétique de changer son composant graphique associé.

6.5 Composants déplaçables (MovableComponent)

La famille d'objets des composants déplaçables n'est pas associée à un paquetage spécifique, mais à plusieurs classes d'autres paquetages.

Cette famille d'objet correspond aux composants pouvant être déplacés et redimensionnés. Concrètement, dans le programme, ce sont les composants qui possèdent une représentation dite « fantôme » (Illustration 36).

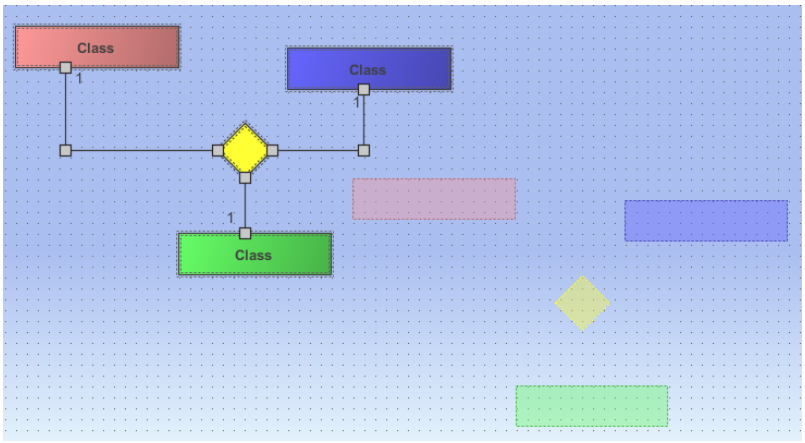


Illustration 36 Exemple des représentations fantômes lors du déplacement de composants graphiques

6.5.1 Structure des composants déplaçables

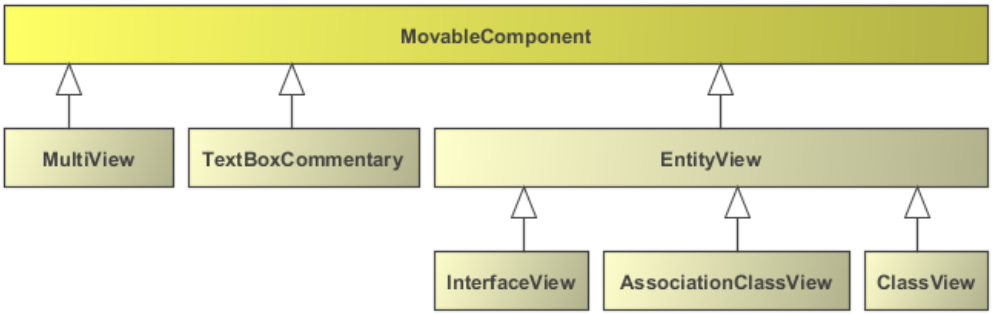


Illustration 37 Structure des composants déplaçables

6.5.2 Fonctionnement des composants déplaçables

Les composants déplaçables forment une famille spéciale de composants du programme car ils agissent de concert. Lorsque l'un deux est déplacé ou redimensionné, il va demander à la vue graphique la liste de tous les composants déplaçables sélectionnés et leurs effectuer le même déplacement / redimensionnement.

Pour accomplir ces actions, les composants déplaçables utilisent les méthodes *move(MouseEvent), apply(MouseEvent), resizeLeft(MouseEvent), resizeRight(MouseEvent)* et *saveMouseLocation(MouseEvent)*.

Informations : L'explication ci-dessous est schématisée à la page suivante.

Pour déplacer ou redimensionner un composant, il faut un point de repère sur lequel se basé pour le calcul du déplacement / redimensionnement. Ce point de repère, c'est la position de la souris lorsque l'utilisateur commence un déplacement / redimensionnement. La méthode *saveMouseLocation(MouseEvent)* permet de faire cela. Quand un composant déplaçable initialise un déplacement / redimensionnement, il va appeler cette méthode pour tous ses semblables sélectionnés.

Ensuite, pour calculer la distance du déplacement / redimensionnement, il suffit de soustraire la position actuelle de la souris à la position enregistrée, puis ajouter le résultat à la position / taille actuelle du composant. Pour cela, les trois méthodes *move(), resizeLeft() et resizeRight()* existent. En fonction de l'action de l'utilisateur, la méthode adéquate de tous les composants sélectionnés sera appelée, avec en paramètre la position actuelle de la souris. Chaque composant va faire son propre déplacement / redimensionnement.

Cependant, les composants affichent une représentation fantôme d'eux-mêmes, leur véritable position / taille n'est pas encore modifiée. Pour confirmer les changements, la méthode *apply()* est finalement appelée, et tous les composants translate leur position et redimensionne leur taille avec celle de leur représentation fantôme.

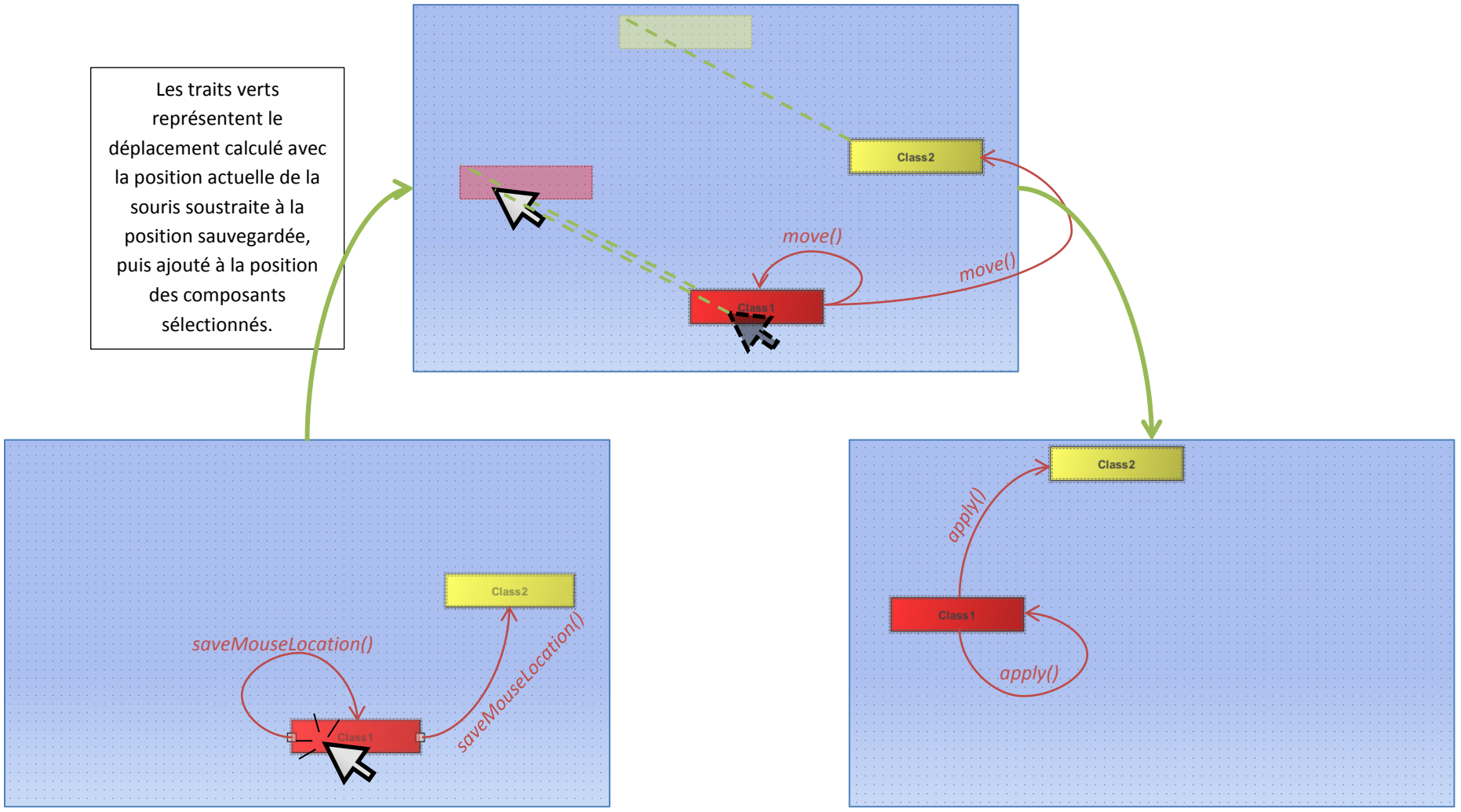


Illustration 38 Mécanismes de déplacement des composant déplaçables

6.5.3 MultiView

Représente simplement une association multiple avec un losange, associées à une multi-association du diagramme de classes UML. Elle possède autant de MultiLineView qu'elle a de rôles. Les MultiLineView sont expliquées au chapitre sur les relations (0).

6.5.4 TextBoxCommentary

Représente une note. Les notes n'utilisent pas de composant Swing pour afficher le texte de leur contenu. Le texte est découpé en mots et chaque mot est dessiné un par un. Lorsqu'un mot dépasse la largeur de la note, il va à la ligne.

Cependant il se peut qu'un mot prenne plus de place que la largeur actuelle de la note. Pour contourner le problème, la taille de la note est calculée puis adaptée dynamiquement chaque fois que la note est redessinée.

6.5.5 Entités (graphic.entity)

La structure graphique des classes et interfaces est contenu dans la classe EntityView. Une EntityView possède une TextBoxEntityName pour l'affichage de son nom, une collection de TextBoxAttribute pour ses attributs et une autre collection de TextBoxMethod pour ses méthodes.

La hauteur de l'entité est calculée en additionnant la taille de toutes ses TextBox et en y ajoutant des marges.

Les TextBox des entités sont les seuls éléments du programme qui ne sont pas directement données à la vue graphique. Elles sont entièrement gérées par l'entité elle-même (ce qui signifie que c'est elle qui dessine et redirige les événements souris sur les TextBox). Comme ces TextBox n'existent pas pour la vue graphique, quand on clique sur une TextBox, c'est comme si l'on cliquait sur une entité. L'avantage vient du fait que l'on peut ainsi déplacer une entité même en cliquant sur sa TextBox. Autrement il aurait fallu, pour chaque TextBox, qu'elle puisse se déplacer et déplacer l'entité à qui elle appartient. Aussi, quand on ouvre le menu contextuel sur une TextBox, c'est celui de l'entité qui s'ouvre.

Ce choix a été fait premièrement pour la raison citée ci-dessus, et aussi car sinon il n'y aurait pas eu beaucoup d'espace disponible pour sélectionner une entitée.

Quand une entité se dessine, elle va automatiquement positionner correctement ses attributs et méthodes en fonction de leur position dans la collection.

6.6 Relations (graphic.relations)

Les relations sont des lignes qui peuvent être uniquement graphique, ou se rapporter à des composants UML comme des agrégations, des compositions ou encore des dépendances.

6.6.1 Structure des relations

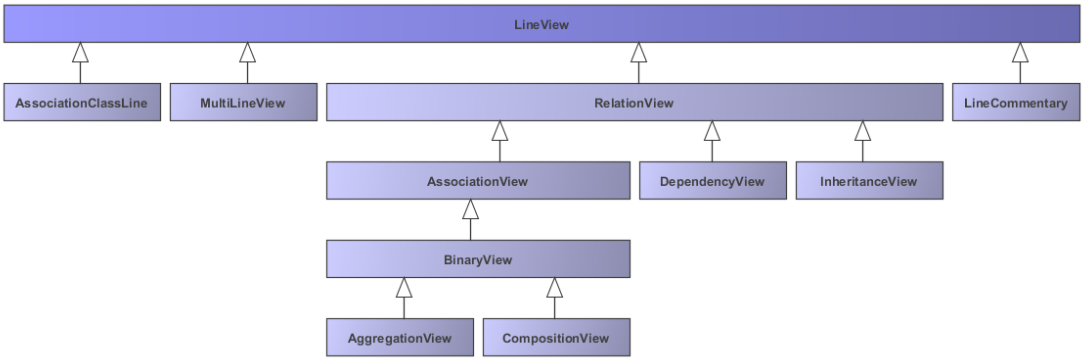


Illustration 39 Structure des relations

Les principales particularités des relations sont leurs carrés blancs (« grips »). Le chapitre sur ces composants explique le fonctionnement des relations (page - 28 -).

Il y a deux familles distinctes de relations, celle purement graphique et celle associée à un élément du diagramme de classe UML.

6.6.1.1 *Relations purement graphique*

Une relation purement graphique signifie qu'elle n'est associée à aucun composant UML. Les relations de l'illustration ci-dessus, représentées en rouge, sont purement graphiques.

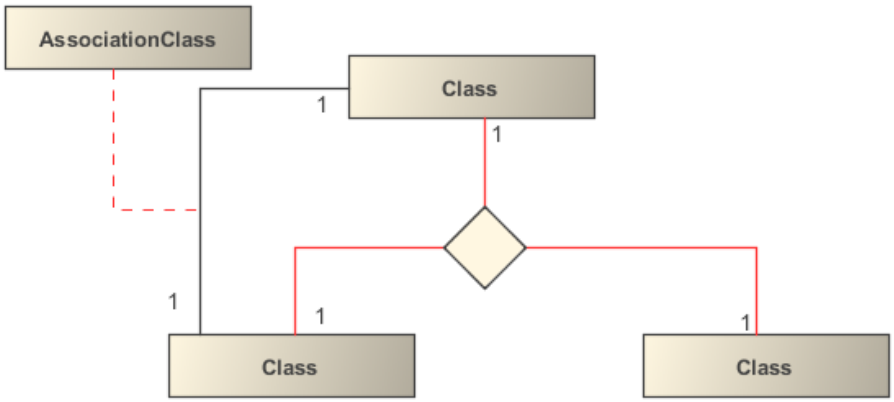


Illustration 40 Relations purement graphique

Malgré que les associations graphiques de la multi-association possèdent un rôle, elles sont purement graphiques. En effet, c'est le composant central (le losange) qui est associé à l'élément UML d'une multi-association et non pas les relations qu'elle affiche.

6.6.1.2 *Relations associées à un composant UML*

Ces relations implémentent l'interface Observer et écoute les changements de leur composant associé. Les relations UML n'étant pas des structures de données complexes (elles ont souvent qu'une entité source et une autre entité cible), leur homologue graphique n'a pas grand-chose à observer si ce n'est l'éventuelle sélection du composant. D'où la simplicité des classes étendant RelationView (chacune ne faisant pas plus d'une dizaine de lignes de code).

6.7 TextBox (graphic.textbox)

Ce package contient un lot d'éléments graphiques, héritant de *GraphicComponent*, permettant d'afficher et de modifier du texte sur une seule ligne. Ce package contient également le composant graphique représentant les notes.

6.7.1 Structure de classes

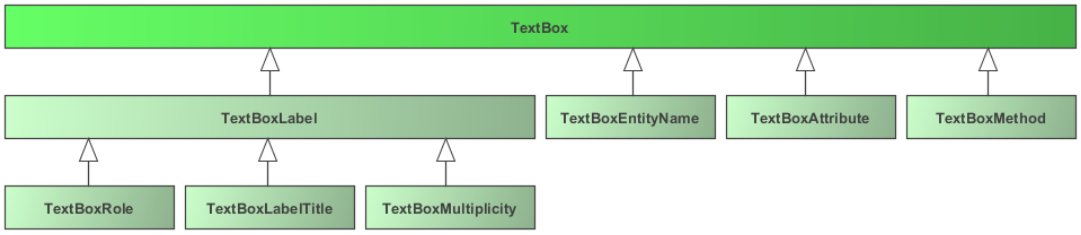


Illustration 41 Structure des classes de la famille TextBox

6.7.1.1 *TextBox*

Une TextBox, dans Slyum, représente une simple zone de texte (Illustration 42). Elles sont facilement reconnaissables dans l'application par son aura grise lorsque le curseur de la souris est placé au-dessus.

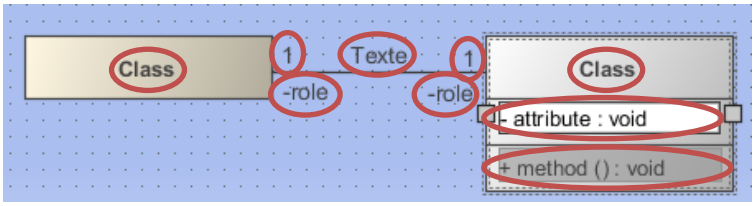


Illustration 42 Exemples de représentation de TextBox

La classe de base, TextBox, permet d'afficher du texte à un endroit donné et de le modifier. Pour modifier le texte d'une TextBox, il suffit d'appeler la méthode *editing()* de la TextBox. Lorsque cette méthode est appelée, un composant Swing JTextEdit va se positionner à l'endroit de la TextBox et contenir le texte retournée par la méthode *getText()*. Lorsque l'utilisateur confirme la modification du texte, la TextBox va modifier son contenu avec la méthode *setText(String)*.

6.7.1.2 *TextBoxEntityName*

Cette TextBox est utilisée pour l'affichage du nom d'une entité (classes, interfaces, classes d'association). Elle possède pour se faire une référence sur l'entité pour obtenir son nom. Elle surcharge la méthode *setText(String)* afin d'avoir un contrôle sur l'édition du texte (contrôle de la syntaxe du nom de l'entité) et de modifier la structure du diagramme de classes lors du changement du nom

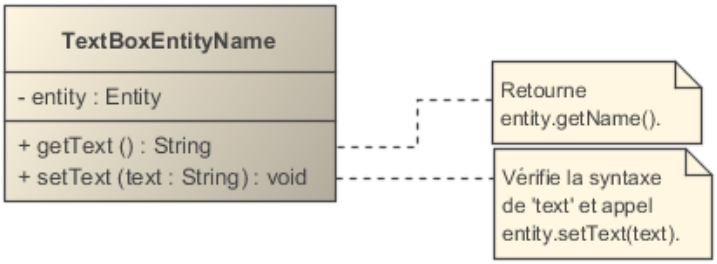


Illustration 43 Classe TextBoxEntityName

6.7.1.3 *TextBoxAttribute & TextBoxMethod*

TextBoxAttribute et TextBoxMethod fonctionnent exactement de la même façon que TextBoxEntityName, excepté le fait qu'elles prennent un attribut, respectivement une méthode, à la place d'une entité. Elles parsent le texte selon la syntaxe présentée au chapitre 2.1 Structures de données.

6.7.1.4 *TextBoxLabel et ses enfants*

Contrairement aux autres TextBox, les TextBoxLabel ont la particularité de pouvoir être déplacées par l'utilisateur en maintenant le bouton de la souris enfoncé. En plus de pouvoir être déplacées, elles possèdent une référence sur un composant graphique. Cette référence permet de dessiner une ligne entre le TextBoxLabel et le composant associé lorsque l'utilisateur survol le TextBoxLabel (Illustration 44).

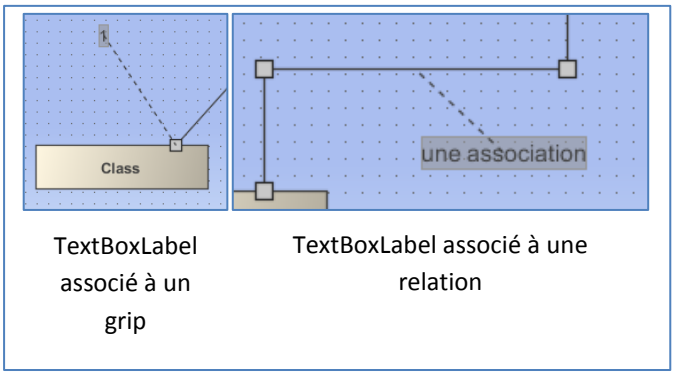


Illustration 44 Représentation de l'association d'un TextBoxLabel

La position d'un TextBoxLabel est relative au composant graphique qui lui est associé. Si le composant graphique se déplace, le TextBoxLabel se déplacera de la même distance.

Les enfants de la classe TextBoxLabel sont des TextBox associées à un composant du diagramme de classe, de la même manière qu'un TextBoxEntityName.

6.8 Création de composants graphiques (graphic.factory)

La création d'un nouveau composant UML (classes, interfaces, relations, …) se fait à l'aide de l'interface utilisateur (Illustration 45).

Illustration 45 Barre d'outils pour l'ajout de composants

Cette barre d'outils fonctionne avec la vue graphique du programme (GraphicView). Pour faire le lien entre l'interface utilisateur et l'ajout d'un nouveau composant au diagramme de classes, nous avons besoin d'une classe intermédiaire ; une fabrique.

Chaque composant graphique possède sa propre fabrique permettant de personnaliser entièrement l'ajout d'un nouveau composant. En effet, les fabriques étendent *GraphicComponent* et sont donc eux aussi des éléments graphiques à part entière. Les fabriques étaient nécessaires car lorsque l'on clique sur un composant de la barre d'outils, celui-ci ne se crée pas tout de suite. Par exemple pour une nouvelle classe, il faut choisir son emplacement. Pour une nouvelle association, il faut sélectionner les deux classes à relier.

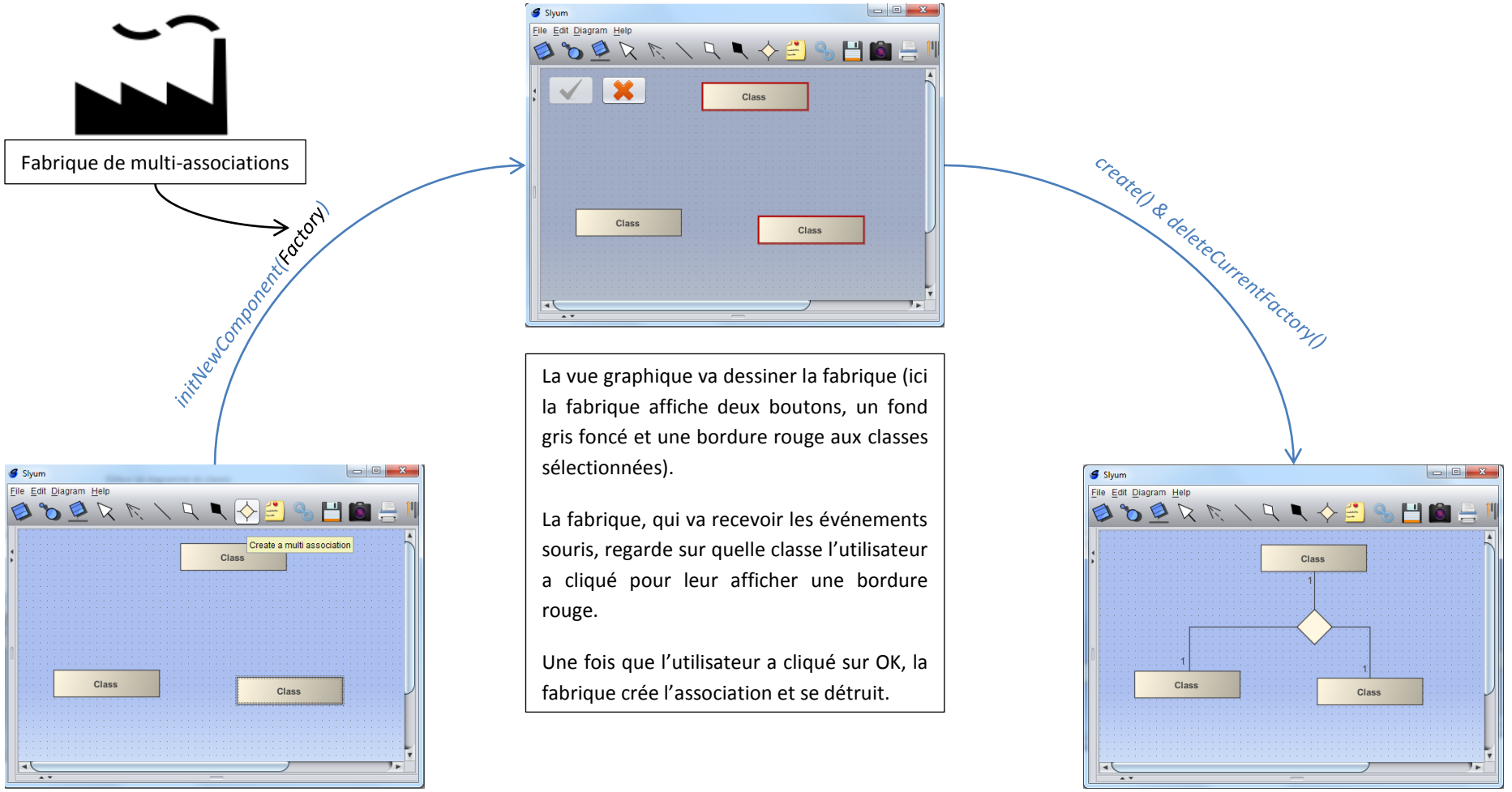
Ainsi, lorsque l'on veut créer un nouveau composant, on va commencer par créer la fabrique qui lui est associé et la donner au *GraphicView*. Le *GraphicView* possède un attribut nommé *currentFactory* qui est à *null* lorsqu'il n'y a pas de nouveau composant en cours. Mais lorsqu'une nouvelle fabrique est passée au *GraphicView*, *currentFactory* la référence et tous les événements souris sont redirigés sur la fabrique au lieu des composants habituels.

De cette manière, la fabrique connaît l'endroit où l'utilisateur clique avec la souris, ainsi que les composants sur lesquels il a cliqué. De plus, comme la fabrique est un composant graphique, elle peut afficher sur le diagramme de classes une représentation intermédiaire du composant prochainement créé.

Une fois que la fabrique a tous les éléments nécessaires à la création du composant auquel elle se rapporte, elle va le créer (méthode *create()*) puis annoncer à la vue graphique (GraphicView) qu'elle a terminée (méthode *deleteCurrentFactory()*). La vue graphique va alors reprendre son exécution normale.

Une fabrique peut très bien décider de ne pas créer de composant s'il survient un problème. Si l'utilisateur tente de créer une association entre une classe et une autre association par exemple. Dans ce cas la fabrique va quand même annoncer à la vue graphique qu'elle a terminé, sans créer de composant.

6.8.1 Exemple de la fabrique des multi-associations



La vue graphique va dessiner la fabrique (ici la fabrique affiche deux boutons, un fond gris foncé et une bordure rouge aux classes sélectionnées).

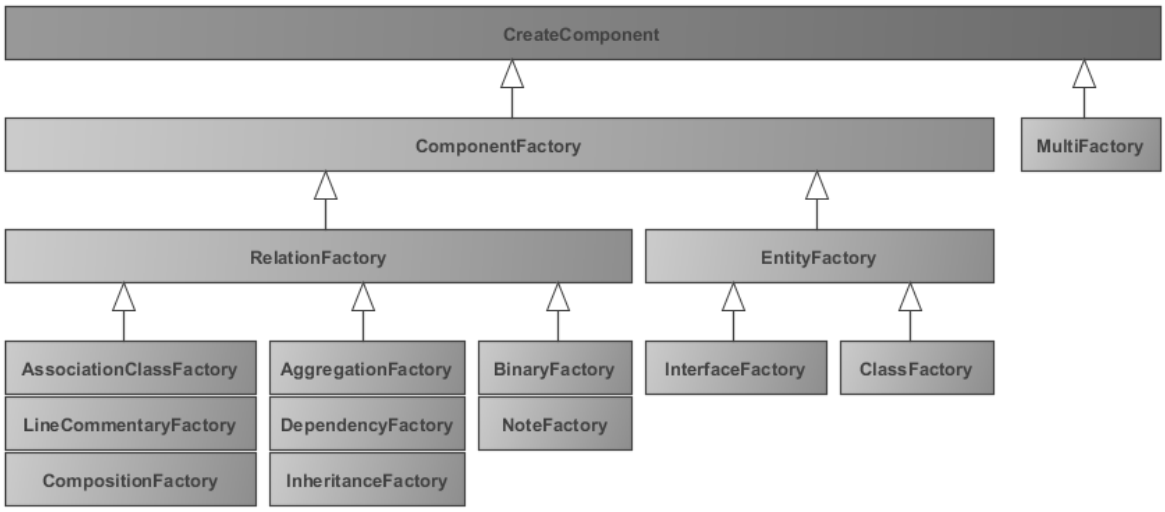
La fabrique, qui va recevoir les événements souris, regarde sur quelle classe l'utilisateur a cliqué pour leur afficher une bordure rouge.

Une fois que l'utilisateur a cliqué sur OK, la fabrique crée l'association et se détruit.

Illustration 46 Fonctionnement des fabriques

6.8.2 Structure des fabriques

Le nom des classes indique le nom des composants que peu créer la fabrique ainsi nommée. Seules les « feuilles » de cette structure permettent de créer des composants.



6.8.2.1 *CreateComponent*

CreateComponent est la classe de base des fabriques. Elle définit les méthodes permettant la suppression de la fabrique (*deleteFactory()*) et la création d'un composant (*create()*).

6.8.2.2 *ComponentFactory*

Les ComponentFactory sont une famille de fabriques créant automatiquement un composant après un click de l'utilisateur. Par click, il faut comprendre un événement gMousePressed suivi d'un événement gMouseReleased. Elle contient quatre attributs ; le composant sur lequel l'utilisateur a cliqué / relâché la souris ainsi que sa position dans ces deux derniers cas. La création du composant est automatiquement appelée à la fin de l'événement gMouseReleased.

Cette fabrique permet de créer la plupart des composants graphiques. Que ce soit juste un click (EntityFactory) comme lors de la création d'une classe ou la désignation de deux composants (RelationFactory) en maintenant la souris enfoncé du premier composant jusqu'au second composant.

6.8.2.3 *MultiFactory*

Cette fabrique spéciale permet de choisir plusieurs classes pour la création d'une multi-association.

7 Interface utilisateur

Mis à part la vue graphique, il y a deux autres vues du diagramme de classes utilisant des composants Swing, ce sont la vue hiérarchique et la vue des propriétés.

7.1 Vue hiérarchique

La vue hiérarchique est une représentation de la structure du diagramme de classes sous forme de liste en arbre. Cette représentation est faite à l'aide du composant Swing *JTree* (Illustration 47) à partir de la structure du système définie par le package *classDiagramm*.

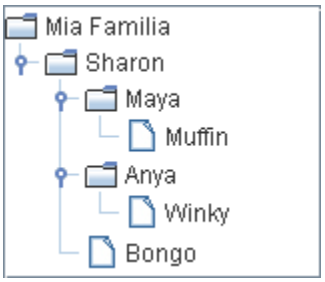


Illustration 47 Composant Swing JTree (http://download.oracle.com)

7.1.1 Implémentation de la vue hiérarchique

Les nœuds de la vue hiérarchique correspondent à un élément du diagramme de classe. Un nœud peut être une classe, une interface ou une relation (association, héritage, dépendance).

Comme les commentaires font intégralement partie de la représentation graphique du diagramme de classe et non de la structure du système, ils ne sont pas affichés dans la vue hiérarchique.

La vue hiérarchique permet deux choses. La première est d'avoir une vue d'ensemble sur tous les composants de la structure du système et de pouvoir les sélectionner, les modifier et les supprimer aisément. La seconde permet, dans le cas de plusieurs vues[2], de dupliquer rapidement les composants du système sur la vue souhaitée.

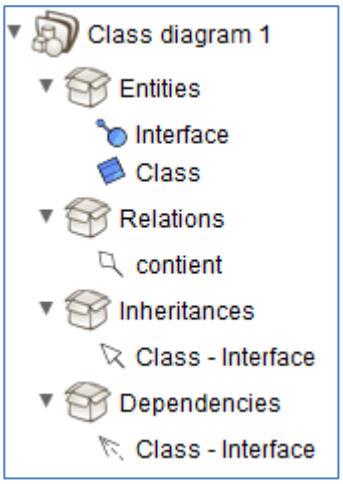


Illustration 48 Représentation de la vue hiérarchique

La vue hiérarchique est décomposée en deux parties ; la première contient les entités (classes et interfaces) et la seconde les relations (nœuds « Entities » et « Relations »). Cette seconde partie est elle-même découpé en 3 parties, représentant les associations, les dépendances et les héritages. Pour éviter une surcroissance de nœuds, toutes les associations (association binaire, n-aire, multi et classe d'association) sont intégrées dans la même branche (« Association »)[3].

Remarques supplémentaires

- Chaque entité peut être dépliée pour afficher son contenu (attributs et méthodes).
- Le contenu de l'arbre est trié selon l'ordre définit dans la structure des classes.

[2] La structure d'un système peut être représentée par plusieurs vues (diagrammes de classes) différentes, mais se référant à la même structure de données.
[3] Une amélioration intéressante serait de séparer une famille relation spécifique dans un sous-nœud si celle-ci serait en grand nombre dans le diagramme de classe.

7.1.2 Classes nécessaires

Les nœuds de la vue hiérarchique, représentant un objet concret du diagramme de classe, doivent pouvoir modifier, et se modifier, suivant les modifications de ce dernier. Pour que les nœuds puissent observer leur correspondant dans le diagramme de classe, ils doivent implémenter l'interface *Observer* et écouter le bon *Observable*. Chaque nœud est étendue de la classe Swing *DefaultMutableTreeNode* en lui ajoutant l'élément structurel (attribut, méthode, nom de classe, etc) adéquat. Ce système fonctionne plus ou moins de la même manière que les *TextBox* du programme.

7.1.3 Structure de la vue hiérarchique

La vue hiérarchique est une vue, au même niveau que le GraphicView. Cela signifie que chaque fois qu'une modification est apportée à la structure du diagramme de classes, celle-ci se répercute sur la vue hiérarchique. Le diagramme ci-dessous montre que la vue hiérarchique implémente IComponentsObserver, l'interface permettant d'observer le diagramme de classes.

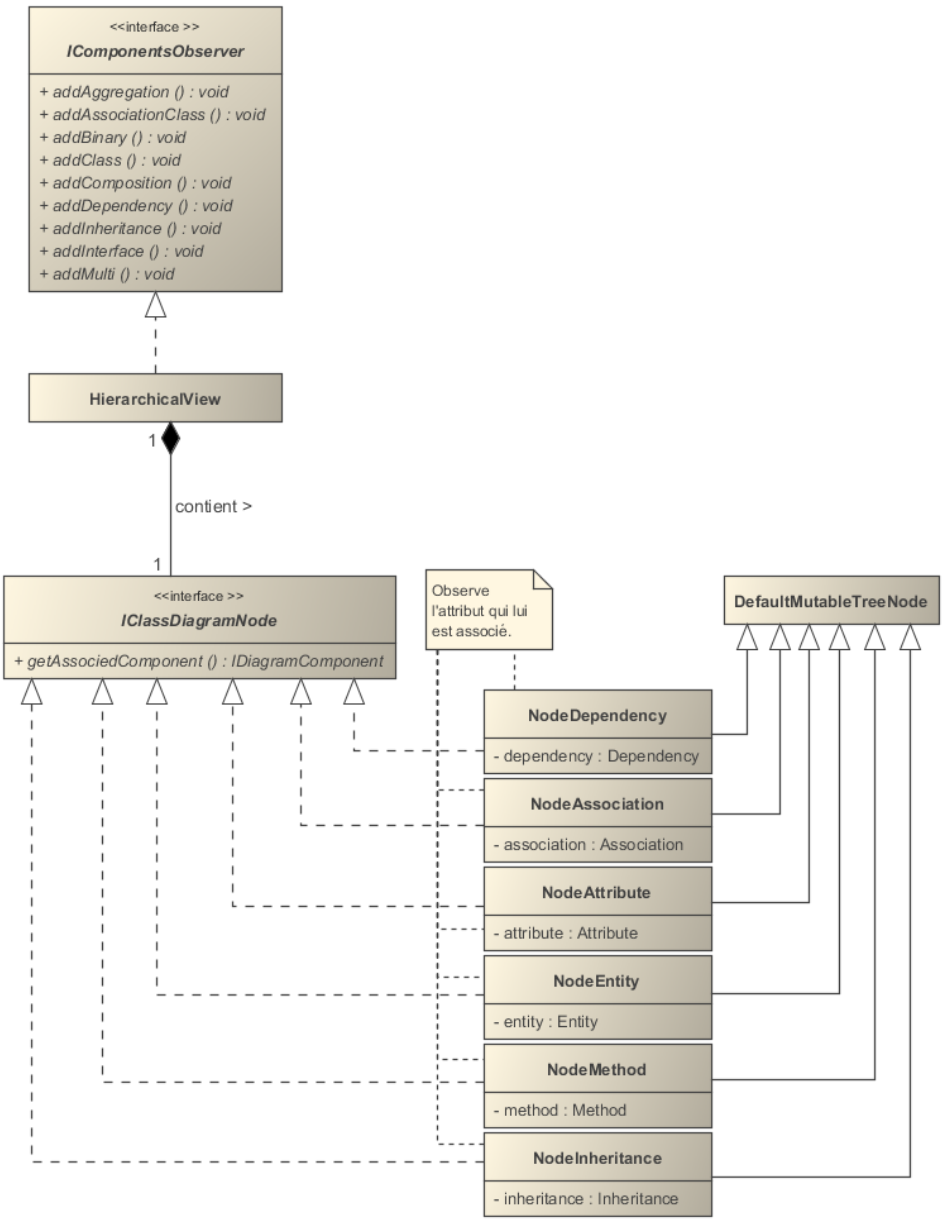


Illustration 49 Structure de la vue hiérarchique

Chaque nœud étend la classe Swing DefaultMutableTreeNode, qui correspond au nœud d'un JTree. Elle implémente également l'interface IClassDiagramNode qui ne signifie rien d'autre que le nœud en question possède un élément associé au diagramme de classe. Tout comme une EntityView, par exemple, NodeEntity est son équivalent pour la vue hiérarchique. Le principe est expliqué au chapitre 5.1 IDiagramComponent.

7.2 Vue des propriétés

La vue des propriétés est une autre vue du programme Slyum. Comme toute vue, celle-ci implémente également l'interface IComponentObserver et écoute le diagramme de classes (classDiagram). La vue des propriétés est différentes des deux autres vues par le fait qu'elle n'affiche qu'un composant à la fois, là où les autres vues affichent l'ensemble des composants du diagramme de classe.

Le diagramme de classes, en plus de notifier les vues lorsqu'un nouveau composant est ajouté ou supprimé, va également envoyer une notification lorsqu'un composant est sélectionné. La vue des propriétés affiche les propriétés du dernier composant sélectionné. Concrètement, les propriétés d'un composant ne sont rien d'autres que des composant Swing dans un JPanel, chaque composant peut posséder ou non une vue pour ses propriétés.

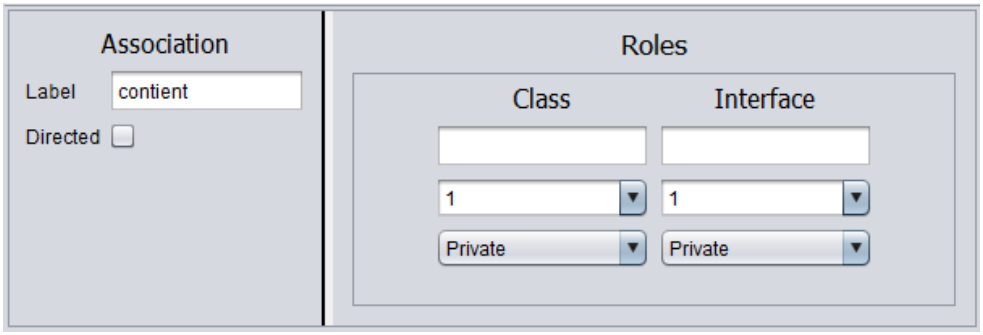
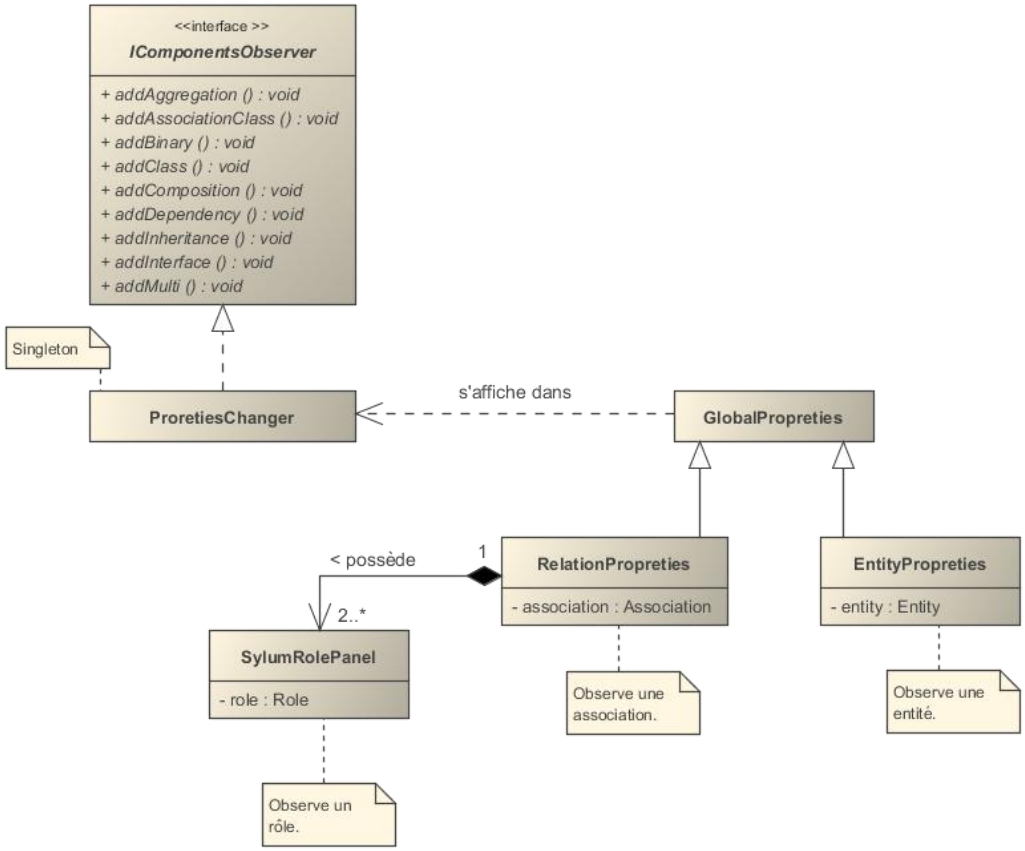


Illustration 50 Vue des propriétés pour une association binaire

7.2.1 Structure de la vue des propriétés



7.2.1.1 *PropretiesChanger*

La classe PropretiesChanger va se charger de faire le lien entre un nouveau composant UML et sa vue des propriétés. Puis, lorsqu'un composant est sélectionné, la vue adéquate sera notifiée et s'affichera dans la vue des propriétés. La classe PropretiesChanger, visuellement, est un JScrollPane situé au sud de l'application. Pour qu'une vue s'affiche dedans, elle va simplement changer son ViewPort.

7.2.1.2 *GlobalPropreties*

C'est la classe parente des vues qui implémente l'interface Observer. Dès qu'elle reçoit une notification de sélection d'un composant, elle va s'afficher dans le ViewPort de la classe PropretiesChanger.

7.2.1.3 *RelationPropreties et EntityPropreties*

Ce sont les classes affichant certaines propriétés pour les composants auxquels elles se rapportent. Par exemple, la vue des propriétés pour les entités (EntityPropreties) possède un JTextFiel pour modifier le nom de l'entité et un JCheckBox pour définir si l'entité est abstraite ou non. Seules ces deux vues existent, mais on pourrait facilement concevoir d'en ajouter d'autres, pour les notes ou pour le diagramme de classe dans sa globalité par exemple.

7.3 Exportation en image

L'exportation en image peut se faire dans un fichier image (PNG, GIF ou JPEG) ou directement dans le presse-papier.

Le premier calcul nécessaire pour l'exportation en image est de connaître la zone contenant les éléments graphiques. Pour connaître cette zone, on prend tous les éléments graphiques contenus dans la vue graphique (GraphicView) pour rechercher la plus petite valeur de la position horizontale et verticale ainsi que la plus grande valeur de la position horizontale et verticale. Ces valeurs permettent de former une zone rectangulaire contenant tous les éléments graphiques. Une marge de 20px est ajoutée à cette zone.

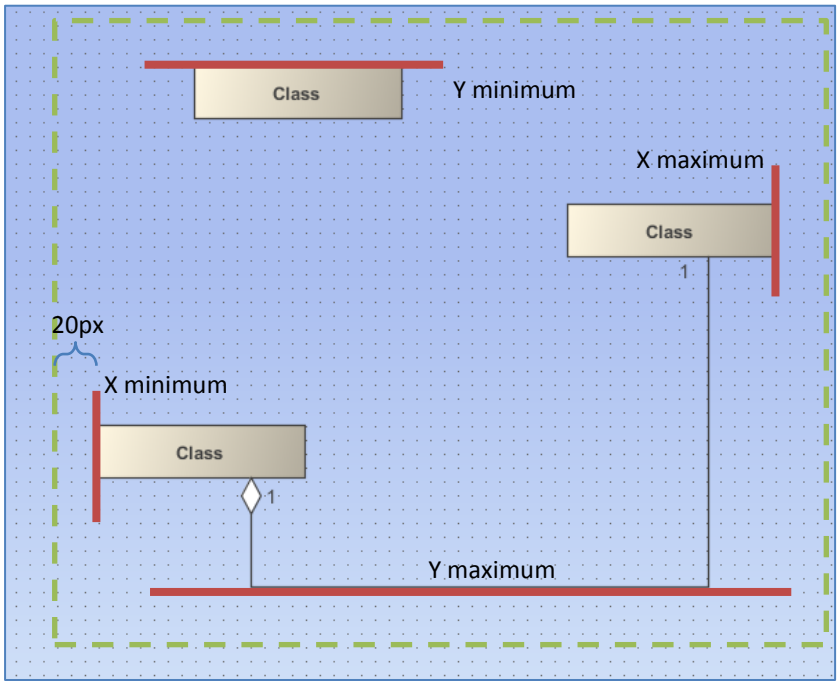


Illustration 51 Zone définie pour l'exportation en image

Ensuite une nouvelle BufferedImage est créé avec cette taille. Puis, au lieu de dessiner la scène sur le JPanel, on va donner le contexte graphique de l'image à la méthode paintComponent() pour dessiner la scène sur cette image. Avant de dessiner la scène, une translation est effectuée pour ramener la scène à l'origine. La translation est de g.*translate*(-(minX + 20), -(minY + 20)).

8 eXtensible Markup Language

Le langage XML (eXtensible Markup Language) est utilisé pour l'enregistrement ainsi que le chargement des diagrammes de classes. Ce langage à l'avantage d'être générique en proposant de créer sa propre structure de données. Ainsi les diagrammes de classes exportés pourront facilement être repris et compris par une personne étrangère au projet. Un fichier binaire (ou un fichier texte sans structure normalisée) n'aurait pas pu être lu (ou difficilement) par une autre application.

La description (structure et types) du schéma XML est faite avec le langage XML-Schema[i]. Ce langage a été choisi par rapport au DTD (Définition de Type de Document[ii]) pour sa plus grande lisibilité ainsi que la possibilité de créer des types de données.

8.1 Description XML

La structure du fichier est fournie en annexe.

La racine du document XML est le nœud « *classDiagram* ». Cette balise contient tout d'abord la structure du diagramme de classes (balise « *diagramElements* ») puis les informations (balise « *umlView* »).

```
<classDiagram>
    <diagramElements>
        <!-- Structure de classes -->
    </diagramElements>
    <umlView>
        <!-- Informations graphiques -->
    </umlView>
</classDiagram>
```

Illustration 52 Structure de base XML

Dans le nœud « *diagramElements* » sont définis les éléments suivants :

- Classes
- Interfaces
- Associations
- Héritages
- Dépendances
- Classes internes

Ils doivent être placés dans l'ordre indiqué et peuvent apparaître autant de fois (ou zéro) qu'il est nécessaire.

La structure de la description XML est découpée en deux parties, en rapport avec le paradigme Modèle-Vue. La première partie, le modèle, contient uniquement les données de la structure du système (les classes, interfaces, relations, etc.). La seconde partie, la vue, se concentre sur les informations graphiques du diagramme de classes (la position (x, y) d'une classe, d'une interface ou d'une relation).

8.1.1 Modèle

Les nœuds suivants correspondent aux informations du diagramme de classes (modèle).

Classes

Une classe est représentée par le nœud « *class* » et possède les attributs suivants :

Nom	Type	Obligatoire ?	Valeur par défaut
id	ID	Oui	-
name	string	Oui	-
visibility	Visibility	Non	« public »
isAbstract	boolean	Non	« false »

Une classe contient les nœuds suivants :

Nom	Type
• method	Operation
• attribute	Variable

Interface

Une interface est représentée par le nœud « *interface* » et possède les attributs suivants :

Nom	Type	Obligatoire ?	Valeur par défaut
id	ID	Oui	-
name	string	Oui	-
visibility	Visibility	Non	« public »

Une interface contient les nœuds suivants :

Nom	Type
• method	Operation

Association

Une association est représentée par le nœud « *association* » et possède les attributs suivants :

Nom	Type	Obligatoire ?	Valeur par défaut
id	ID	Oui	-
name	string	Oui	-
direction	boolean	Non	« false »
aggregation	Aggregation	non	« none »

La direction définit si l'association est bidirectionnelle ou non. En mettant la valeur à « false », l'association sera bidirectionnelle.

Une association contient les nœuds suivants :

Nom	Type
• role	Role
• associationClass	Class

Si l'association possède une classe d'association, celle-ci est définie dans le nœud « *associationClass* ». Si l'association n'a pas de classe d'association, le nœud peut être ignoré.

Le nœud « *role* » doit apparaître au moins deux fois.

Héritage

Une relation d'héritage est représentée par le nœud « *inheritance* » et possède les attributs suivants :

Nom	Type	Obligatoire ?	Valeur par défaut
id	ID	Oui	-

Une relation d'héritage contient les nœuds suivants :

Nom	Type
• child	int
• parent	int

Les deux nœuds « *child* » et « *parent* » sont obligatoires, ne peuvent apparaître qu'une seule fois et contiennent l'id de la classe à laquelle ils se rapportent.

Dépendance

Une relation de dépendance est représentée par le nœud « *dependency*» et possède la même structure qu'une relation d'héritage, excepté le nom de ses deux nœuds qui sont « *source* » et « *target* », à la place de « *child* » et « *parent* » de la relation d'héritage.

Classe interne

Une classe interne est représentée par le nœud « *innerClass* » et possède la même structure qu'une relation d'héritage ou de dépendance. La seule différence réside dans le nom de ses deux nœuds qui sont cette fois « *boundingClass* » et « *innerClass* ». Le nœud « *boundingClass* » correspond à l'id de la classe englobante et « *innerClass* » à la classe interne.

Visibilité

Ce type d'énumération (*Visibility*) correspond à la visibilité d'une classe, méthode ou d'un attribut et peut prendre les valeurs suivantes : « *public* », « *protected* », « *private* » ou « *package* ».

Agrégation

Ce type d'énumération (*Aggregation*) permet de définir le type d'une association. Il peut être soit « *none* » qui représente une association simple, « *compose* » ou « *aggregate* » qui représentent, respectivement, une association de composition ou d'agrégation.

Variable

Ce type représente une variable utilisée pour un attribut de classe ou un paramètre d'une opération. Il possède les attributs suivants :

Nom	Type	Obligatoire ?	Valeur par défaut
name	string	Oui	-
type	string	Oui	-
const	boolean	Non	« false »
visibility	Visibility	Non	-
defaultValue	string	Non	-
collection	int	Non	0
isStatic	boolean	Non	false

L'attribut « *const* » définit si la variable est constante ou non et « *collection* » si c'est un tableau et sa taille (0 veut dire que ce n'est pas un tableau).

Certains de ces attributs (visibility, collection, isStatic), suivant le langage de programmation, n'existent pas comme paramètres d'une opération. Ils sont tout simplement ignorés lors de l'importation du fichier XML.

Opération

Une opération est représentée par le nœud « *operation* » et possède les attributs suivants :

Nom	Type	Obligatoire ?	Valeur par défaut
name	string	Oui	-
returnType	string	Oui	-
visibility	Visibility	Non	« public »
isStatic	boolean	Non	« false »
isAbstract	boolean	Non	« false »

Une opération contient le nœud suivant :

Nom	Type
• parameter	Variable

Rôle

Les rôles d'une association correspondent aux classes auxquelles l'association se réfère. Ils contienent le nom de la variable du type de la classe, sa visibilité ainsi que sa multiplicité (la première association de l'Illustration 4 comporte les deux rôles *variable1* et *variable2*). Un rôle est représenté par le nœud « *role* » et possède les attributs suivants :

Nom	Type	Obligatoire ?	Valeur par défaut
name	string	Oui	-
visibility	Visibility	Non	« public »
componentId	int	Oui	-

L'attribut « *componentID* » définit l'id du composant (classe ou interface) auquel le rôle se réfère.

Rôle - Multiplicity

Un rôle contient un nœud « *multiplicity* ». Ce nœud définit la multiplicité du rôle, la structure du nœud est la suivante :

```
<multiplicity>
    <min>int</min>
    <max>int</max>
</multiplicity>
```

int doit être remplacé par la multiplicité minimum, respectivement maximum du rôle.

8.1.2 Vue

Les nœuds suivants correspondent aux nœuds contenant les informations graphiques.

Cette partie définit deux types qui sont « *Geometry* » et « *Line* ». Le nœud racine de la vue est « *UmlView* » et contient les nœuds suivants (qui peuvent apparaître autant de fois que nécessaire, dans l'ordre indiqué) : « *componentView* » puis « *relationView* ».

Geometry

Le type « *Geometry* » représente un rectangle contenant quatre informations ; la position x et y du coin supérieur gauche du rectangle par rapport au bord du conteneur parent ainsi que la longueur et la hauteur du rectangle (Illustration 53).

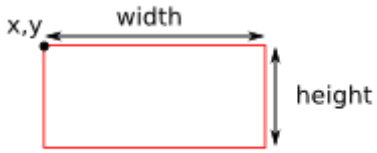


Illustration 53 Coordonnées d'un rectangle (http://www.liafa.jussieu.fr)

Un nœud de type « *Geometry* » à la structure suivante :

```
<geometry>
    <x>int</x>
    <y>int</y>
    <w>int</w> <!-- width -->
    <h>int</h> <!-- height -->
</geometry>
```

Line

Un type « *line* » définit une collection de points qui permettent de définir le tracer de plusieurs lignes. Le nœud doit avoir au minimum deux points. La représentation d'une ligne se fait en traçant un segment du premier point au second, du second au troisième et ainsi de suite jusqu'au dernier point.

Un nœud de type « *Line* » à la structure suivante :

```
<line>
    <point>
        <x>int</x>
        <y>int</y>
    </point>
    <point>
        <x>int</x>
        <y>int</y>
    </point>
    <!-- ... -->
</line>
```

componentView

Un nœud « *ComponentView* » contient un seul attribut :

Nom	Type	Obligatoire ?	Valeur par défaut
componentId	int	Oui	-

Cet attribut permet d'identifier le composant auquel il se rapporte. Un nœud « *ComponentView* » est utilisé pour sauvegarder les coordonnées d'une classe ou d'une interface. Elle contient alors un nœud de type « *Geometry* » :

Nom	Type
• geometry	Geometry

relationView

Un nœud « *RelationView* » contient aussi un seul attribut qui définit la relation à laquelle les coordonnées du nœud se rapportent :

Nom	Type	Obligatoire ?	Valeur par défaut
componentId	int	Oui	-

Les coordonnées de la relation sont enregistrées à l'aide d'un nœud de type « *Line* » :

Nom	Type
• line	Line

8.2 Exemple

Voici un exemple du code XML (uniquement le modèle) pour le diagramme suivant :

```
<classDiagram>
    <diagramElements>
        <class id="0" name="Animal" isAbstract="true">
            <method name="manger" returnType="void" isAbstract="true">
                <parameter name="nourriture" type="Nourriture" />
            </method>
            <attribute name="age" type="int" visibility="private"/>
        </class>
        <class id="1" name="Chien" />
        <class id="2" name="Poile" />

        <association id="3" aggregation="aggregate">
            <role componentId="2">
                <multiplicity>
                    <min>0</min>
                    <max>Infinity</max>
                </multiplicity>
            </role>
            <role componentId="1">
                <multiplicity>
                    <min>1</min>
                    <max>1</max>
                </multiplicity>
            </role>
        </association>

        <inheritance id="4">
            <child>1</child>
            <parent>0</parent>
        </inheritance>
    </diagramElements>
    <umlView>
        <!-- Informations non disponibles -->
    </umlView>
</classDiagram>
```

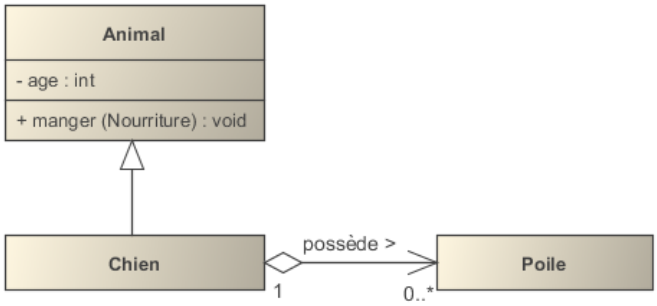


Illustration 54 Correspondance Diagramme de classes - XML

9 Conclusion

Voici le bilan du développement de l'éditeur de diagramme de classe Slyum.

Le projet a été mené à bien et, en l'état, est utilisable pour la création de diagrammes de classes. La création de tous les composants UML définis dans le cahier des charges est possible à l'exception des classes internes qui ne sont pas encore implémentées. Ces composants correspondent aux classes, interfaces, classes d'association, notes, relations d'héritages et de dépendances, associations binaires, n-aire (multi-association), composition, agrégation, rôles, multiplicités, attributs et méthodes.

Les actions élémentaires de personnalisation du diagramme de classes sont également intégrées comme l'ajout, la suppression, la sélection, le redimensionnement ou encore le déplacement des composants.

Un certain nombre de paramétrages sont également disponibles comme le changement de couleur, la disposition des composants (superposition), ordre des attributs et des méthodes, composants visibles d'une classe (tout, seulement les attributs, seulement les méthodes, rien), le type d'affichage des paramètres d'une méthode (tout, seulement le type, seulement le nom, rien) ou encore le positionnement et l'ajustement automatique des composants graphiques.

Une vue sous forme d'arbre hiérarchique est également intégrée. Cependant, il était prévu de pouvoir directement agir sur les composants du diagramme de classes depuis cette vue (ajout, suppression et modification). Ces fonctionnalités n'ont pas encore été implémentées.

Une autre vue détaillant les propriétés des composants (vue des propriétés) est disponible pour les classes (y compris leurs attributs et méthodes) et les associations. Cette vue devrait être ajoutée pour tous les composants graphiques existants à l'avenir.

Pour ce qui est de l'exportation du diagramme de classes. L'exportation en image est terminée, ainsi que la possibilité de copier les composants sélectionnés dans le presse-papier sous format PNG. Il est également possible d'imprimer un diagramme de classe. Cependant, les marges et autres paramètres de mise en page ne sont pas encore implémentés. La fonctionnalité d'impression est, pour l'instant, difficilement utilisable sans l'affichage des marges.

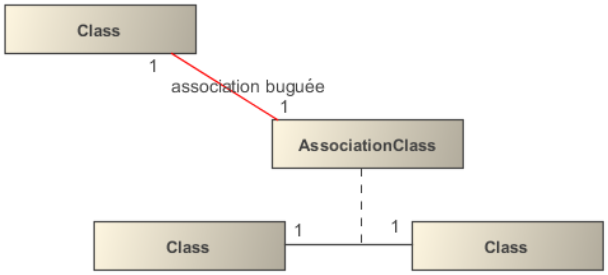
Enfin, l'enregistrement en format XML est principalement terminée, mais pas totalement. Pour rappel, le format XML (fichier texte) est le format utilisé pour enregistrer les diagrammes de classes. Malheureusement, celui-ci n'est pas entièrement terminé et donc le chargement de diagramme de classe peut ne pas représenter fidèlement celui enregistré. Cependant, ce sont uniquement des informations graphiques qui manquent, vous ne perdrez donc aucun composant lors de l'enregistrement / chargement

Pour finir, en utilisant le programme dans la pratique, on remarque qu'un certain nombre de modifications, d'ajustements, ou d'ajouts mineurs serait appréciable pour le confort et la vitesse d'utilisation du programme (voir Prochains ajouts, au chapitre suivant). Malgré ces quelques faiblesses, le programme est totalement utilisable en l'état et permet de créer et personnaliser des diagrammes de classes avec tous les principaux composants du langage UML (la majeure partie des diagrammes de classes de ce rapport ont d'ailleurs été fait avec Slyum (le temps ne me permettant pas de tous les refaire)).

9.1 Problèmes connus

Voici la liste des problèmes connus du programme.

- Lorsque l'on fait une relation (quelle qu'elle soit) sur une classe d'association et que l'on enregistre le projet, il est impossible de charger le projet par la suite. Cela est dû au fait que les classes d'association sont chargées avant les associations, et que lorsqu'elle cherche une association, celle-ci n'existe pas encore. Le seul moyen de l'éviter est de ne pas faire cette manipulation.



- Les colonnes « Type » des JTable de la vue des propriétés ne peuvent pas être éditées avant qu'il y a eu au moins une modification du type en question depuis une autre vue. Source du problème inconnue.
- Lorsque l'on modifie une information depuis la vue des propriétés, celle-ci n'est pas soumise à la syntaxe adéquate. Par exemple, depuis la vue des propriétés, il est possible de mettre des caractères illégaux dans le nom d'une classe.
- Les caractères spéciaux n'ont pas été encodés dans le fichier XML et lors du parsing, le parser SAX utilisé génère des exceptions. Il ne faut pas mettre, où que ce soit (notes, nom de classe, nom d'attributs, …), des caractères spéciaux tant que ce problème n'est pas corrigé, sinon vous ne pourrez plus charger le projet.
- Les rôles ne se chargent pas ! La multiplicité ainsi que le nom du rôle n'est pas chargé correctement et se réinitialise lors du chargement du fichier (il est cependant correctement enregistré).
- Le changement des classes participant à une multi-association n'est pas répercuté sur le diagramme de classes. De plus, lier une note à un élément graphique ne s'enregistrera pas.

9.2 Fonctionnalités non implémentées

Cette liste expose les implémentations urgentes (outre la correction des problèmes listés ci-dessus) et requises avant de pouvoir utiliser Slyum confortablement…

- Enregistrement du type de vue des paramètres des méthodes (Nom et Type, Type, Nom, Rien).
- La position des multi-associations n'est pas enregistrée.
- La position de la ligne entre une classe d'association et une association n'est pas enregistrée.
- Il n'est pas possible de faire des collections de types pour les types.
- Faire le chargement du diagramme de classe sur un thread séparé.
- Ajouter la possibilité de modifier la structure des classes depuis la vue hiérarchique.
- La position des TextBox pour les rôles, label et multiplicité n'est pas sauvegardée.
- La validation du diagramme de classe n'est pas implémentée (interdiction de manipulations impossibles comme une relation d'héritage A -> B et B -> A).
- Ajout de différents messages de confirmation (comme lors de la suppression de composants).
- Ajouter la possibilité de lier des classes à une association multiple existante.

9.3 Prochains ajouts

Cette liste indique différentes améliorations intéressantes pouvant être intégrées prochainement.

- Possibilité de changer l'ordre des attributs / méthodes depuis la vue des propriétés
- Copier et coller certains éléments graphiques.
- Possibilité d'annuler / rétablir une modification (Ctrl + Z)
- Changer la taille de la bordure des classes
- Possibilité de copier les méthodes héritées ou implémentées par une classe lors de la création d'une relation d'héritage
- Compléter les menus déjà existants et en ajouter d'autres
- Améliorer la gestion des types (en proposant ceux déjà existants dans une liste déroulante)
- Ajouter des labels aux multi-associations ?
- Mieux gérer la sauvegarde des projets en détectant s'il a subit une modification avant de demander sa sauvegarde ou sa fermeture.
- Permettre d'exporter le diagramme de classes dans un langage de programmation (Java).
- Permettre de créer un diagramme de classe à partir d'un code (Java) (reverse engineering).

10Liste des références

- http://laurent-audibert.developpez.com/Cours-UML/html/Cours-UML016.html
- http://www.visual-paradigm.com/VPGallery/diagrams/Class.html#class_enumeration
- http://www.info.fundp.ac.be/~jml/TP_AMSI/TP6/MetaModeleClassDiagram.pdf
- http://web.univ-pau.fr/~ecariou/cours/idm/cours-meta.pdf
- http://www.commentcamarche.net/contents/xml/xmldtd.php3
- http://zvon.developpez.com/tutoriels/dtd/
- http://www.learn-xml-schema-tutorial.com/Simple-Type-Elements.cfm
- http://java.developpez.com/faq/xml/?page=sax
- http://nadeausoftware.com/node/82#Addinggradientstoauserinterface
- http://www.liafa.jussieu.fr/~carton/Enseignement/InterfacesGraphiques/MasterInfo/Cours/Swing/graphiques.html
- http://download.oracle.com/javase/tutorial/ui/features/components.html
- http://download.oracle.com/javase/tutorial/uiswing/components/toolbar.html
- http://java.sun.com/products/jfc/tsc/articles/painting/
- http://download.oracle.com/javase/tutorial/uiswing/components/menu.html#popup
- http://download.oracle.com/javase/tutorial/uiswing/misc/focus.html

11Table des illustrations

12Journal de travail

Semaine 1

- ✓ Faire un méta-schéma d'un diagramme de classes UML.

Dimanche, le 20 février 2011

- Création du diagramme de classe au format numérique (avec Enterprise Architect 8.0) pour la modélisation des diagrammes de classes (méta-schéma).
- Lecture de documentations sur les diagrammes UML, notamment sur les relations entre les classes.
- Lecture du tutoriel[iii] sur l'apprentissage des bases d'utilisation de Swing.
- Création d'un modèle pour les documents avec page de garde et index.
- Début de la rédaction du journal de travail et du cahier des charges.

Semaine 2

- ❖ Correction du diagramme de classes. Il manque les relations d'héritages, les rôles, les associations multiples et les classes d'associations. De plus, la distinction du type de relation doit être faite à l'aide de classes et non pas d'un type énuméré.

Mardi, le 22 février 2011

Modifications du diagramme UML pour correspondre aux exigences voulues lors de la séance du 21 février. J'ai recommencé la modélisation du diagramme avec le programme ArgoUML pour tester ses fonctionnalités par rapport à Enterprise Architect. Le diagramme maintenant corrigé a été envoyé par e-mail.

- Début de la rédaction du cahier des charges.

Vendredi, le 25 février 2011

- Rendez-vous exceptionnel avant la semaine de relâches pour mettre au point le diagramme UML.

Semaine de relâches

Jeudi, le 3 mars 2011

- Il faut faire un test du diagramme de classe en faisant la structure en Java, avec Eclipse. Le programme devra afficher une représentation du schéma sous forme textuelle (à l'aide de phrases) afin d'effectuer des tests pour voir si tous les éléments voulus des diagrammes de classes peuvent être représentés par notre diagramme de classe. Commencement de cette application.
- Continuation du cahier des charges, ajout du chapitre sur les spécificités d'UML 1.4.

Vendredi, le 4 mars 2011

- Ajout du reste des classes pour l'application de test ainsi que les attributs et les méthodes d'accès.
- Mise au propre du diagramme de classe fait lors du rendez-vous du 25 février 2011 sur Enterprise Architect (non terminé).
- Fin du chapitre sur les spécificités du cahier des charges. Ajout du chapitre sur les fonctionnalités demandées. Il reste à mettre les illustrations.

Semaine 3

- ✓ Terminer la rédaction du cahier des charges.
- ✓ Terminer le programme de test en Java.

- ❖ Il ne faut pas mettre d'illustrations dans le cahier des charges.
- ❖ La sortie du programme de test doit maintenant être au format XML plutôt que textuelle.

Mardi, le 8 mars 2011

- Lecture de documentations sur XML et XMI. Création de la définition de type de document (DTD) pour l'exportation au format XML.
- Continuation de la programmation de l'application Java. Ajout des classes représentant les relations (associations binaires, agrégation et composition, multi-associations).
- Mise au propre (format numérique) du diagramme de classe fait lors de la semaine 2.

Samedi, le 12 mars 2011

- Ajout des méthodes « toString » dans chacune des classes afin de permettre leur exportation au format XML respectant la DTD mise en place.
- Ajout de deux sous-classes à la classe « Inheritance », « Generalize » et « Realize » afin de permettre un contrôle plus strict sur les paramètres du constructeur de la classe « Inheritance ». Une interface ne doit pas pouvoir avoir comme parent une classe.
- Implémentation de la classe « Role ».

Dimanche, le 13 mars 2011

- Test du programme java avec un diagramme de test et exportation de ce diagramme sous format XML.
- Vérification du fichier XML de sortie. Correction de certains problèmes qui ne correspondaient pas à la syntaxe UML ainsi qu'à la DTD. Le document de sorti est maintenant bien formé.
- Modification du nom des méthodes « toString » par « toXML », ajout du paramètre deep afin de rendre possible la mise en page avec des tabulations.
- Réduction significative du cahier des charges. La version complète sera conservée pour le rapport final. La version courte sera entrée dans GAPS.
- Il manque les classes d'associations. Ajout des classes d'associations.

Semaine 4

- ✓ Ajouter un diagramme de classe pour la version graphique.
- ✓ Ajouter un XSD pour l'exportation au format UML graphique.
- ✓ Pouvoir importer un schéma XML, pas uniquement l'exporter.
- ✓ S'informer sur les éventuelles bibliothèques disponibles pour le dessin du diagramme de classe du programme.

- ❖ Modifier la DTD par un XSD (XML Schema Definition).
- ❖ Modifier le format XML, mettre l'héritage directement dans les classes.
- ❖ Mettre au propre le diagramme de classe.
- ❖ Une classe d'association peut être attribuée à une multi-association (pas uniquement à une association binaire).

Lundi, le 14 mars 2011

- Mise au propre du diagramme de classe.

Jeudi, le 17 mars 2011

- Apprentissage du format XSD pour les spécifications du format XML ainsi que transformation du DTD en XSD.
- Corrections de différents éléments de la DTD lors du passage à la XSD. Suppression de « BinaryAssociation », « MultiAssociation » et « AssociationClass » pour les fusionner en une seule « Association », la distinction se fera grâce aux attributs et nombre de rôles. Suppression également de l'élément « Inheritance » pour l'intégrer directement en tant qu'élément « parent » dans les « class » et « interface ».

Samedi, le 19 mars 2011

- Création de deux versions du diagramme de classe en ajoutant les informations nécessaires au positionnement graphique des composants. Une version intègre les informations graphiques directement dans les classes, une autre version les intègres dans des classes spécialisées pour la vue.
- Création de deux versions de schémas XML (XSD). Les deux versions ajoutent les informations pour dessiner les composants. La première ajoute ces informations directement dans les éléments, la seconde les places à la fin, dans un ensemble de balises prévu pour.

Dimanche, le 20 mars 2011

- Recherche d'informations sur des bibliothèques graphiques en Java pour la représentation du diagramme de classe. Parmi celles trouvées, deux semblent intéressantes : Piccolo2D et GLG. La première semble parfaitement adaptée aux besoins de l'application, est gratuite et facile d'utilisation. La seconde est plus professionnelle et est déclinée en une version gratuite et une payante. Elle semble en outre destinée aux professionnels et moins adaptée pour ce travail.

Semaine 5

- ✓ Se familiariser avec la bibliothèque Swing de Java.

- ❖ Effectuer différentes corrections vues lors de la séance (héritage, compléter diagramme de classe).
- ❖ Chercher une autre bibliothèque graphique que Piccolo2D, qui n'est plus mis à jour depuis mars 2010.
- ❖ Se familiariser avec la bibliothèque Swing de Java.

Jeudi, le 24 mars 2011

- Recherche d'une nouvelle bibliothèque graphique. JHotDraw 7 semble convenir pour le remplacement de Piccolo2D. Les fonctionnalités sont les mêmes et la dernière version date de janvier 2011. En outre, la licence est LGPL.
- Correction du diagramme de classe et du fichier xsd.

Semaine 6

- ✓ Créer les méthodes permettant d'exporter le graphique au format xml ainsi que celle permettant son importation.

- ❖ Compléter le diagramme de classe selon la correction faite lors de la séance. Cela correspond aux points suivants.
- ❖ Ajout des références.
- ❖ Ajout des classes internes.
- ❖ Ajout des collections.
- ❖ Ajout des propriétés (levé d'exceptions par exemple).
- ❖ Enlever les stéréotypes.
- ❖ Changer la structure du document xsd pour la rendre plus lisible.

Mardi, le 5 avril 2011

- Corrections du fichier xsd.
- Ajout du package « view » et de ses classes aux projets. Correction des méthodes « toXML() » du projet.
- Correction du diagramme de classe.

Mercredi, le 6 avril 2011

- Ajout des méthode « toXML() » pour les vues.

Semaine 7

- ✓ Regarder la bibliothèque swt d'IBM.
- ✓ Finir l'exportation et faire l'importation XML.
- ✓ Commencer l'interface graphique.

Mardi, le 12 avril 2011

- Création du projet Slyum en testant la bibliothèque SWT de IBM pour remplacer Swing. Après quelques tests, SWT sera utilisée pour le travail de bachelor.
- Début de l'interface graphique utilisateur, création du panneau latéral gauche contenant la vue d'une classe, de ses attributs et méthodes (sobrement nommée SpeedClassViewer).

Mercredi, le 13 avril 2011

- Ajout de la classe « AttributeProperties » permettant de lister les attributs d'une classe dans une liste. Ajout également du « PanelAttributChanger », une boîte de dialogue flottante permettant l'édition rapide d'un attribut lors d'un clic sur celui-ci.

Semaine 8

- ✓ Finir l'exportation et faire l'importation XML.
- ✓ Continuer l'interface graphique.

Jeudi, le 14 avril 2011

- Mise en place d'un repository sur google code.
- Création d'un logo pour le programme.
- Corrections et améliorations diverses pour le « SpeedClassViewer ».

Vendredi, le 15 avril 2011

- Création de la fenêtre permettant d'ajouter de nouvelles méthodes, avec un nombre de paramètre dynamique.

Samedi, le 16 avril 2011

- Ajout de la barre d'outils contenant les éléments pour ajouter des classes, interfaces et relations. Création des icônes de classe et interface pour cette barre d'outils.

Lundi, le 18 avril 2011

- Installation d'une machine virtuelle avec Ubuntu 10.10 et Mac Os 10.4 afin d'avoir une représentation de l'interface graphique sur ces différents OS, et éventuellement les changements à effectuer.
- Sur Ubuntu, le programme n'a besoin d'aucune modification notable.
- Pour Mac OS X, certains boutons ne sont pas affichés dans leur intégralité. Analyse de solutions à mettre en place.

Semaine 9

- ✓ Continuer l'interface graphique.

Mardi, le 26 avril 2011

- Création du GraphicsView, le composant permettant d'afficher, déplacer et modifier le diagramme de classe.
- Création de la vue d'une classe.

Mercredi, le 27 avril 2011

- Possibilité de déplacer et de redimensionner une classe.
- Ajout de l'interdiction de faire sortir la classe du GraphicsView.
- Ajout d'une grille et du magnétisme.
- Ajout d'un fond dégradé pour la GraphicsView et les classes.

Jeudi, le 28 avril 2011

- Possibilité d'ajouter des classes en cliquant sur le bouton adéquat dans la barre d'outils.
- Changement de la façon de placer les classes sur la grille. Avant, une méthode était appelée avant de changer la position et la taille, maintenant, les méthodes qui changent la taille et la position sont directement redéfinie et place automatiquement la classe sur la grille.

Vendredi, le 29 avril 2011

- Modification de l'aspect visuel des classes lors du survole de la souris et de la sélection de celle-ci.
- Ajout de la sélection et de la sélection multiple.

Semaine 10

- ✓ Création de la visualisation des classes.

Mardi, le 3 mai 2011

- Changement de la méthode de dessin. Utilisation d'images au lieu de Canvas. Cela demandera de redéfinir tous les évènements des objets, mais permet aussi une plus grande flexibilité, notamment pour l'exportation du diagramme en image.

Semaine 11

- ✓ Continuer la représentation graphique.
- ✓ Faire un diagramme de classe sur la structure du GraphicsView, les composants utilisés pour dessiner le diagramme de classe.

Vendredi, le 13 mai 2011

- Continuer la représentation avec la nouvelle méthode de dessin. Ajout de la sélection multiple et de l'aspect d'une classe sélectionné.
- Ajout d'une représentation « fantôme » qui s'affiche quand une classe est déplacée. Cela permet de mettre à jour le diagramme uniquement quand l'utilisateur relâche le bouton de la souris lors d'un déplacement.

Samedi, le 14 mai 2011

- Ajout de la possibilité de redimensionner les classes sélectionnées.
- Ajout d'une taille minimum pour le redimensionnement d'une classe.
- Ajout de la représentation des attributs d'une classe.

Dimanche, le 15 mai 2011

- Possibilité d'ajouter des attributs en cliquant sur un bouton.
- Ajout d'un effet de survol de la souris sur un attribut et le nom de la classe.
- Possibilité de modifier un attribut en double cliquant dessus.
- Possibilité d'ajouter un attribut grâce au bouton « A » d'une classe.
- Possibilité de supprimer un attribut grâce au bouton « X » situé au niveau de l'attribut.

Semaine 12

- ✓ Refonte de l'interface graphique.
- ✓ Faire un diagramme de classe pour les relations.

Mardi, le 17 mai 2011

- Refonte de l'interface graphique. Suppression du panneau de gauche pour le remplacer par une zone où les propriétés de l'élément sélectionné seront affichées.
- Possibilité de sélectionner une classe, un attribut ou une méthode.

Dimanche, le 22 mai 2011

- Création du diagramme de classes pour les relations.
- Implémentation de la structure des classes pour les relations.

Semaine 13

- ✓ Recommencer le projet avec Swing.
- ✓ Faire un diagramme de classe pour la vue graphique.

Mardi, le 24 mai 2011

- Suite à des problèmes de ralentissement sur la version SWT (beaucoup de flicking), je décide, après avoir effectué des tests, de recommencer le projet sur Swing, qui est plus rapide.
- Création d'une structure de classes plus solide pour le passage sur Swing. La version SWT n'était pas aboutie, ni optimisée.
- Création de la grille, du fond ainsi que d'une barre d'outils pour ajouter de nouveau composants.

Mercredi, le 25 mai 2011

- Création de la représentation graphique d'une classe, ainsi que sa fabrique.
- Création d'un effet de sélection sur une classe.
- Création des carrés blancs pour le redimensionnement des classes.

Vendredi, le 27 mai 2011

- Ajout de la possibilité de redimensionner une ou plusieurs classes.

Samedi, le 28 mai 2011

- Ajout de la possibilité de déplacer une ou plusieurs classes. Avec une représentation « fantôme » (le composant ne se déplace pas, mais un rectangle transparent le représente jusqu'à ce que l'utilisateur confirme le déplacement).

Dimanche, le 29 mai 2011

- Création de la fabrique pour créer une relation (plus précisément une relation d'héritage). La fabrique affiche une flèche blanche suivant la souris jusqu'à ce que l'utilisateur confirme la nouvelle relation.

Semaine 14

- ✓ Continuer de ré-implémenter les composants avec Swing.
- ✓ Améliorer la personnalisation des relations.

Mardi, le 31 mai 2011

- Création de la relation d'héritage.
- Création des ancrages aux bords d'une classe pour que la relation ne passe jamais au-dessus des classes qu'elles relient.
- Mise à jour de la disposition des relations lorsque les classes sont déplacées.
- Possibilité de déplacer une relation avec la souris.
- Possibilité de personnaliser une relation en maintenant Ctrl+Gauche sur la relation. Un nouvel ancrage apparaît, pouvant être déplacé et définissant le chemin de la relation.

Vendredi, le 3 juin 2011

- Possibilité d'ajouter des interfaces ainsi que de l'affichage d'un stéréotype. Le stéréotype pour l'interface est par défaut « interface ».
- Ajout de la relation de réalisation. Si une relation d'héritage est faite entre une classe et une interface, ou une interface et une autre interface, la relation est automatiquement représentée en tant que réalisation (« realize »).

Samedi, le 4 juin 2011

- Ajout d'un menu contextuel sur les classes et interfaces.
- Ajout d'une nouvelle classe (« TextBox ») permettant d'afficher du texte et de le modifier via une JTextBox. Elle sera utilisée pour l'affichage du nom d'une classe, pour les méthodes et attributs ainsi que les rôles et le titre d'une relation.
- Possibilité d'ajouter de nouvelles méthodes à une classe ou une interface depuis le menu contextuel correspondant.

Dimanche, le 5 juin 2011

- Possibilité d'ajouter de nouveaux attributs à une classe depuis le menu contextuel de la classe.

Semaine 15

- ✓ Mettre au propre le rapport intermédiaire.
- ✓ Changer la méthode pour ajouter de nouvelles intersections aux relations.
- ✓ Mettre en italique le nom d'une interface.
- ✓ Supprimer les intersections de relations inutiles automatiquement.

Mercredi, le 8 juin 2011

- Modification de la méthode d'ajout d'une intersection à une relation. Avant il fallait cliquer sur la touche « Contrôle » + Clique gauche et maintenant il suffit de cliquer.
- Ajout de la possibilité d'éditer le nom d'une classe / interface et le nom des interfaces apparaissent maintenant en italique.
- Les intersections des relations sont automatiquement supprimées si elle ne change pas (ou très peu) la direction de la relation.

Dimanche, le 12 juin 2011

- Écriture du rapport intermédiaire.

Lundi, le 13 juin 2011

- Écriture du rapport intermédiaire.

Mardi, le 14 juin 2011

- Écriture et mise au propre du rapport intermédiaire.

Vendredi, le 17 juin 2011

- Présentation et rendu du rapport repoussé au 20 juin 2011.

Dimanche, le 19 juin 2011

- Relecture du rapport.

Semaine 17

- ✓ Début du travail à plein sur le diplôme.
- ✓ Ajout des relations de dépendances et associations.

Lundi, le 20 juin 2011

- Rendez-vous pour une démonstration intermédiaire ainsi que le rendu du rapport intermédiaire.

Mardi, le 21 jui 2011

- Création des dépendances. Possibilité de déplacer et de modifier le contenu du label.
- Ajout d'un aspect visuel permet d'identifier à quel relation appartient un label (un trait relie les deux composants lors du survol de la souris).
- Ajout des rôles.
- Ajout des relations d'agrégations et d'associations.

Mercredi, le 22 juin 2011

- Ajout des notes.

Semaine 18

- ✓ Commencer l'implémentation de l'interface graphique.
- ✓ Ajouter une liaison pour les notes.
- ✓ Implémenter la vue hiérarchique.

Lundi, le 27 juin 2011

- Implémentation de la vue hiérarchique.

Mardi, le 28 juin 2011

- Implémentation des nœuds pour la vue hiérarchique.

Mercredi, le 29 juin 2011

- Ajout de la vue des propriétés.
- Ajout de l'exportation au format image (png, jpg et gif).

Jeudi, le 30 juin 2011

- Ajout de la possibilité d'imprimer le diagramme de classe.
- Ajout de la barre de menus.

Vendredi, le 1 juillet 2011

- Ajout de la boîte de dialogue « Au sujet de… ».
- Amélioration de la vue hiérarchique.

Semaine 19

- ✓ Continuer l'implémentation de l'interface graphique.
- ✓ Continuer l'implémentation de la vue hiérarchique.
- ✓ Ajouter une liaison pour les notes.
- ✓ Continuer l'implémentation de la vue des propriétés.
- ✓ Ajouter le paramétrage de l'application (les couleurs et la taille des bordures, entre autre).

Lundi, le 4 juillet 2011

- Possibilité de sélectionner plusieurs composants dans la vue hiérarchique.
- Amélioration du fonctionnement et de la structure de la vue hiérarchique.

Mardi, le 5 juillet 2011

- Création de la vue des propriétés en bas de l'application pour les classes et interfaces. La vue permet de modifier les attributs, méthodes et paramètres de la classe / interface.

Mercredi, Jeudi et vendredi, les 6, 7 et 8 juillet 2011

- Continuation de l'implémentation de la vue des propriétés.

Semaine 20

- ✓ Ajouter les différents points du document « Prochains ajout.txt »
- ✓ Ajouter l'exportation et l'importation XML.
- ✓ Améliorer la gestion des relations lorsque l'on veut faire une récursivité ou changer la source/cible d'une relation.
- ✓ Ajouter les classes d'association et les multi-associations.
- ✓ Ajouter la possibilité de supprimer les composants.

Lundi, le 11 juillet 2011

- Corrections et ajouts à la structure du fichier XML pour l'exportation.
- Ajout de la possibilité de sauvegarder un le diagramme de classe (uniquement la structure, pas encore la vue).

Mardi, le 12 juillet 2011

- Ajout de la possibilité de modifier une relation en la déplaçant sur une autre classe / interface.
- Correction de différents bugs lors du déplacement d'une relation.
- Préparation de la structure pour l'intégration prochaine des classes d'association et des multi-associations.
- Ajout des classes d'association. Elles peuvent être ajouté soit en choisissant deux classes, soit en sélectionnant une association déjà existante.

Mercredi, le 13 juillet 2011

- Création de l'association multiple, sa fabrique et les relations correspondantes.
- Amélioration du système d'observation entre les grips.
- Ajout d'une partie de la vue dans l'enregistrement (les classes et interfaces).
- Début de la création du parseur XML. Possibilité de charger des classes, interfaces dans le diagramme et dans les vues.

Jeudi, le 14 juillet 2011

- Travail sur la sauvegarde et l'ouverture d'un fichier XML. Création d'une structure objet permettant de reproduire la structure du fichier XML, pour ensuite créer le diagramme de classe et ses vues correspondantes.
- Ajout de l'option pour créer un nouveau projet, qui remplace le projet actuel.
- Ajout des icônes pour les classes d'associations, les multi-associations, l'option du nouveau projet et l'option d'ouverture d'un projet.
- Amélioration de la gestion de la suppression d'un composant graphique pour préparer le terrain pour l'ajout de la fonctionnalité de suppression d'un composant (notamment lors des réactions en chaînes provoquées par la suppression d'un composant (l'effet de la suppression d'une classe sur les relations dont elle participe par exemple).

Vendredi, le 15 juillet 2011

- Ajout de la possibilité de redimensionner la taille du texte.
- Ajout de la possibilité d'ajuster automatiquement la largeur des classes et interfaces.
- Ajout du menu contextuel pour tous les composants graphiques.
- Mise au point de l'affichage des attributs et méthodes, ils sont maintenant correctement au centre de la classe / interface.
- Ajout de la possibilité de supprimer n'importe quel composant du diagramme de classe avec la souris ou le clavier.
- Possibilité de sélectionner tous les composants avec le raccourci CTRL+A.
- Il est maintenant impossible de faire sortir une classe ou une interface du champ de vision.
- Ajout de barres de défilements verticales et horizontales à la représentation graphique du diagramme de classe pour afficher les éléments hors du champ de vision.

Dimanche, le 17 juillet 2011

- Possibilité de changer l'ordre d'affichage des classes et interfaces (agissant sur leur superposition).
- Possibilité de modifier l'ordre des attributs et opérations.
- La vue des propriétés peut maintenant être modifiée et agit sur les éléments correspondants du diagramme de classe.
- Le changement de la vue d'une entité (voir que les attributs, méthodes, rien ou tout) s'applique maintenant à toutes les classes sélectionnées.
- On peut maintenant afficher ou non la grille, ainsi que choisir sa taille parmi un jeu de taille prédéfinie.
- Lorsque l'on déplace deux entités qui ont entre-elles des relations, les relations concernées se déplacement également. Avant, uniquement les deux extrémités de la relation se déplaçaient, rendant pénible le déplacement de plusieurs éléments sans devoir repositionner manuellement chaque relations.
- Ajout de la possibilité de modifier la couleur de fond ainsi que la couleur de base des classes (la couleur attribuée lors de la création d'une nouvelle classe).

Semaine 21

- ✓ Terminer l'implémentation des fichiers de sauvegarde XML.
- ✓ Ajouter la vue des propriétés pour les relations.
- ✓ Ajouter les paramètres des opérations dans la vue des propriétés.
- ✓ Continuer l'implémentation du changement de couleur.
- ✓ Terminer les menus.
- ✓ Rédaction du rapport et d'un mode d'emploi.
- ✓ Ajouter un contrôle sur les actions de l'utilisateur (ex : héritages A->B, B->A impossible).
- ✓ Faire différents ajustement.

Lundi, le 18 juillet 2011

- Test du programme sur Mac OS X. La taille des icônes de la vue hiérarchique est mal adaptée, la couleur des boutons de la fenêtre de propriété ne s'affiche pas, la fabrique de multi-associations se redessine sur elle-même (régler le problème en redessinant entièrement le JPanel au lieu d'une partie spécifique) et la couleur de fond de la barre de menu et d'outils ne se dessinent pas correctement. Le reste fonctionne.

Mardi, le 19 juillet 2011

- Ajout de la possibilité de modifier individuellement, ou seulement les composants sélectionnés, leur couleur.
- Ajout de la possibilité de modifier le style d'affichage des méthodes ; soit voir leur type et leur nom, soit uniquement leur type, seulement leur nom, soit aucun des deux. Ces paramètres peuvent être appliqués individuellement pour chaque méthode, ou pour toutes les méthodes des classes sélectionnées.
- Création de la vue des propriétés pour les relations.

Mercredi, le 20 juillet 2011

- Continuer l'implémentation de la vue des propriétés pour les relations.
- Ajout de la possibilité de sélectionner les multi-associations et d'autres composants.
- Ajout de raccourcis clavier pour accepter ou annuler la création d'une multi-association.

Jeudi, le 21 juillet 2011

- Refonte totale des commentaires. Ils n'utilisent plus de JTextArea et sont des objets de type « ComponentGraphic » à part entière. Ils apparaissent maintenant correctement lors de l'exportation en image.

Vendredi, le 22 juillet 2011

- Création de la classe abstraite « MovableComponent ». Cette classe permet de déplacer les composants en affichant lors du déplacement une représentation « fantôme » du composant. Elle est pour l'instant étendue par « EntityView », « MultiView », et « TextBoxCommentary ».
- Ajout de l'enregistrement des couleurs, des notes, des relations entre les notes et les composants et des paramètres de la vue graphique (taille du texte, taille de la grille et couleur de fond).
- Rédaction du manuel utilisateur.

Samedi, le 23 juillet 2011

- Rédaction du manuel d'utilisation.

Dimanche, le 24 juillet 2011

- Rédaction du manuel d'utilisation.
- Écriture des en-têtes de méthodes et des commentaires du projet.

Lundi, le 25 juillet 2011

- Écriture des en-têtes de méthodes et des commentaires du projet.
- Création d'un plan pour la structure du rapport.

Mardi, le 26 juillet 2011

- Rédaction du rapport et création des diagrammes de classes avec Slyum.
- Ajout de la possibilité de de copier dans le presse papier une image avec tous les composants sélectionnés. Le raccourci clavier est CTRL+C.
- Correction d'une erreur qui empêchait une classe de se redessiner correctement.

Mercredi, le 27 juillet 2011

- Rédaction du rapport.

Jeudi, le 28 juillet 2011

- Ajout du manuel utilisateur au programme.
- Correction d'un problème qui affichait tous les grips lors du chargement d'un projet.
- Rédaction des commentaires de l'interface graphique.
- Ajout de la possibilité d'ajouter et d'éditer les paramètres d'une méthode depuis la vue des propriétés.
- Faire une version temporaire pour Mac OS X.
- Relecture de la documentaion.
- Création du DVD.
- Impression des documents.

13Annexes

- DVD contenant l'ensemble du projet et de la documentation
- Mode d'emploi

[i] Plus d'informations sur la structure XML-Schema : http://www.learn-xml-schema-tutorial.com/Simple-Type-Elements.cfm

[ii] Plus d'informations sur les DTD : http://www.commentcamarche.net/contents/xml/xmldtd.php3 et http://zvon.developpez.com/tutoriels/dtd/

[iii] Tutoriel du site du zéro disponible à l'adresse suivante : http://www.siteduzero.com/tutoriel-3-10444-votre-premiere-fenetre.html