Université Paul Sabatier – Toulouse III
IUT A - Toulouse Rangueil
Projet tuteuré #20

Antoine de Roquemaurel
Mathieu Soum
Geoffroy Subias
Marie-Ly Tang
Groupe B

Pour Monsieur Thierry Millan (Client)
Madame Caroline Kross (Tutrice)

Installation et exploitation

Bibliothèque d'objets graphiques UML

Toulouse, le 3 décembre 2012

Avant-propos

Présentation du projet

LibUML est une bibliothèque d'objets graphiques représentant les différents éléments de modélisation de la norme UML 2[1]. Celle-ci a été développé dans le cadre des projets tuteurés à l'IUT[2] 'A' de Toulouse. Nous l'avons développée en Java et conçue comme une bibliothèque pouvant être utilisée dans des programmes Java comme composant. Vous pouvez vous en servir pour développer un outil complet de modélisation UML par exemple.

Cette bibliothèque permet de faire différentes choses dans le cadre de la conception UML, une fois que vous aurez compris le fonctionnement de libUML vous pourrez :

- Créer des éléments de modélisation
  - Acteur actif ou passif
  - Traitement
  - Cas d'utilisation
  - Classe
- Relier ses composants via différents types de flèches
  - Agrégation
  - Composition
  - Association binavigable ou mononavigable
  - Messages synchrones ou asynchrones
  - Généralisation
- Supprimer des éléments et flèches
- Modifier le contenu des éléments et flèches
- Redimensionner les éléments de modélisation

Ce document a pour but de vous présenter le fonctionnement de la bibliothèque UML et de son démonstrateur.

Présentation du groupe

Notre groupe projet est composé de quatre étudiants de deuxième année de DUT Informatique à l'IUT 'A' de Toulouse, voici la composition de l'équipe :

- Antoine de Roquemaurel
- Mathieu Soum
- Geoffroy Subias
- Marie-Ly Tang

1. Unified Modelling Language
2. Institut Universitaire de Technologies

Téléchargements

Nous vous avons envoyé deux archives zip par email, une contenant le projet Netbeans de la bibliothèque et du démonstrateur et une contenant la bibliothèque.

Bibliothèque et démonstrateur

La première archive est donc un projet netbeans, cette archive contient notre bibliothèque et ses tests associés, le démonstrauteur mais également la bibliothèque `JGraphx` qui est indispensable au bon fonctionnement de notre bibliothèque. Nous ne garantissons le bon fonctionnement de la bibliothèque qu'avec la version de `JGraphx` 1.8, celle-ci n'ayant pas été testée avec des versions différentes.

Le projet netbeans de notre bibliothèque est également disponible en ligne, si vous souhaitez la télécharger de nouveau, elle est donc disponible à l'adresse suivante :

▷ http://telechargements.joohoo.fr/libUML/libUML-netbeans.zip
▷ http://telechargements.joohoo.fr/libUML/JGraphX-1.8.jar

Documentation

Toute la documentation du projet est sous format HTML[3], celle-ci ayant été générée grâce à Javadoc.

Si dans ce manuel nous parlons d'une méthode et que vous n'en comprenez pas l'intéret, vous pouvez aller la chercher dans la documentation, pour chaque classe, le fonctionnement global de la classe est expliquée, l'utilité de chaque attribut, de chaque méthode, et ce que vous devez mettre dans les différents paramètres des méthodes.

Nous vous avons remis un .zip par email contenant quatre dossiers de documentation :
- La documentation privée de la *bibliothèque* (Toutes les méthodes et tous les attributs)
- La documentation publique de la *bibliothèque* (Seuls les méthodes et attributs publics ou protégés)
- La documentation privée du *démonstrateur* (toutes les méthodes du démonstrateur)
- La documentation des *tests unitaires* (Toutes les méthodes de tests)

Celle ci est également disponible en ligne aux adresses suivantes :

▷ http://documentation.joohoo.fr/libUML/bibliothequePrivee/index.html
▷ http://documentation.joohoo.fr/libUML/bibliothequePublique/index.html
▷ http://documentation.joohoo.fr/libUML/demonstrateurPrivee/index.html
▷ http://documentation.joohoo.fr/libUML/testsUnitaires/index.html

Vous pouvez également accéder à la documentation de `JGraphX`, cela peut vous être utile dans certains cas.

▷ http://documentation.joohoo.fr/JGraphX/index.html

3. HyperText Markup Language

Fonctionnement du document

Ce document est un document expliquant notre approche pour développer une bibliothèque d'objets graphiques UML.

Dans ce dossier, vous pourrez repérer diverses notations, cette partie a pour but de vous expliquer les notations afin que vous puissiez lire en toute sérénité.

Le glossaire

Un mot dans le glossaire a une police particulière, vous pourrez savoir qu'un mot est dans le glossaire lorsque vous repérerez un mot avec la police suivante : leMotDansLeGlossiare. Si vous voyez cette police, vous pouvez donc vous référez à l'annexe A page 13.

Les noms de méthode, d'attribut ou de classe

Les mots se référant à un nom présent dans du code ont une police particulière, une police type "machine à écrire", si vous voyez la police suivante, c'est que c'est un nom de méthode, d'attribut ou de classe : `uneFonction`.

Les noms de paquetage

Les noms de paquetage utiliseront une police particulière, afin que l'on puisse les différencier d'une classe ou d'une méthode, vous les trouverez comme suit : unPaquetage.

Les notes de bas de page

Nous utilisons régulièrement des notes de bas de pages, pour donner un acronyme, pour expliquer plus en détail une notion, ces notes de bas de pages sont un numéro en exposant, vous trouverez la note correspondante en bas de la page courante, comme ceci [4].

Les liens hypertextes

Dans le document, nous pouvons faire référence à un lien d'un site web, tous les liens seront donc symbolisés par une petite puce, et une police particulière comme ceci :
▷ `http://monLien.fr/index.html`
Une liste de tous les liens présents dans le document est accessible en annexe B page 14.

4. Ceci est une note de bas de page

Table des matières

Chapitre 1

Le démonstrateur

1.1 Interface Homme Machine

Nous avons développé un démonstrateur afin que vous puissiez comprendre comment fonctionne la bibliothèque, et pour montrer ses possibilités. Ainsi, tout ce qu'il est possible de faire avec la bibliothèque sera présent dans le démonstrateur.

Dans cette partie nous allons passer brièvement ses fonctionnalités, cependant, pour une meilleure compréhension, nous avons commenté le code du démonstrateur pour qu'il soit le plus simple possible et pour que vous ayez le moins besoin de vous référez à ce document. La documentation est également disponible au format javadoc comme signalé en Avant-propos page ii.

Le démonstrateur est volontairement simple, il n'a pas pour but d'être lourd, mais uniquement de montrer les possibilités de la bibliothèque. Il est disposé en trois parties :

1. En haut, la barre d'outils, permettant de sélectionner l'élément graphique souhaité
2. Au centre, ce que nous appelons le graphe, c'est ici qu'apparaîtrons les diagrammes UML
3. À droite, un panneau contenant éventuellement un tableau avec les informations de la classe sélectionnée, mais aussi le type de diagramme choisit

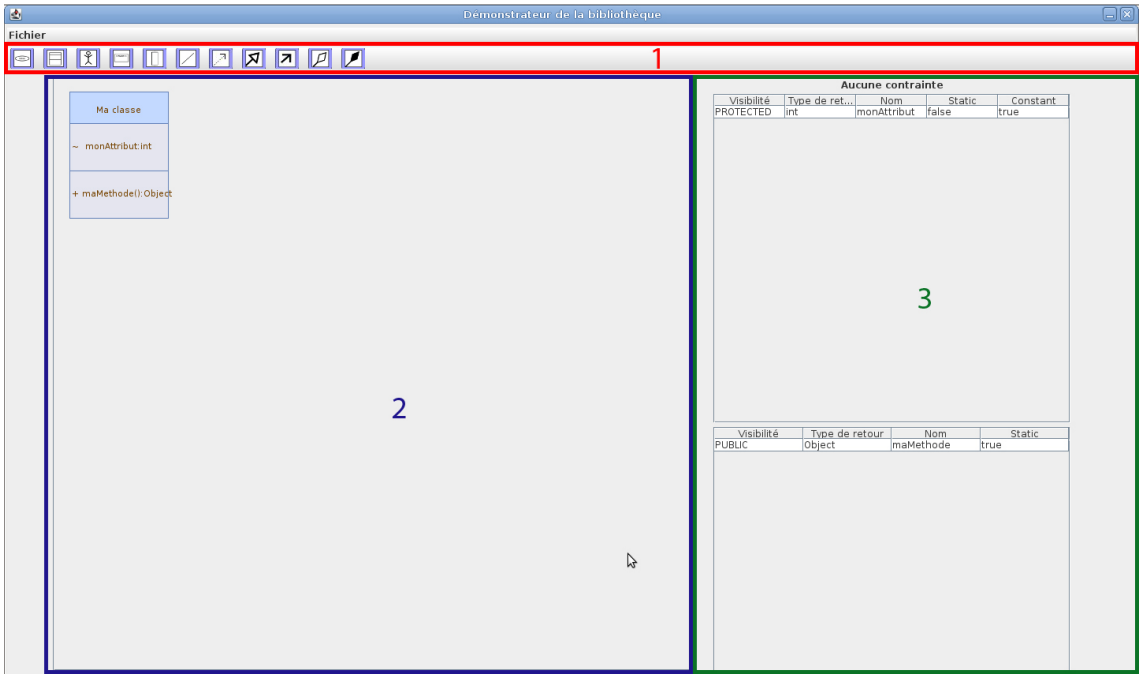


FIGURE 1.1 – Les différentes parties du démonstrateur

Au démarrage du démonstrateur, il faut choisir le type de contrainte souhaitée, cela aura pour effet d'interdire l'ajout de certains éléments.

1.1.1 La barre d'outils

La barre d'outils contient tous les éléments graphiques que notre bibliothèque permet d'introduire.

1.1.1.1 Élément de modélisation

Il est actuellement possible de faire les éléments de modélisation suivants :
- Acteur actif
- Acteur passif
- Traitement
- Cas d'utilisation
- Classe
- Interface

Le clic sur un bouton positionnera un élément de modélisation dans le graphe.

1.1.1.2 Liens

La deuxième partie de la barre d'outils contient tous les liens qu'il est possible de faire.
- Association
- Dépendance fonctionnelle
- Généralisation
- Agrégation
- Composition

Pour relier deux éléments, vous devez cliquer sur la flèche voulue, puis cliquer sur les deux éléments à relier. Le clic sur les deux éléments aura pour effet de créer un lien.
Attention, pour les traitements et les classes, la sélection de l'élément se fait par le haut de l'élément.

1.1.2 Le graphe

Le graphe contient tous les éléments graphiques, l'affichage d'un élément se fera donc dans le graphe. C'est également dans le graphe que s'affiche le menu contextuel permettant d'interagir avec les éléments.

Le graphe est également important dans la programmation comme nous le verrons section 2.2 page 6.

1.1.3 Le menu contextuel

Le menu contextuel s'affiche au clic droit sur un élément graphique, celui-ci diffère quelque peut suivant l'élément.

1.1.3.1 Les acteurs (actifs ou passifs)

Il est possible d'afficher, ou non la ligne de vie, celle ci étant utile pour les diagrammes de séquence.

1.1.3.2 Les classes

Il vous est possible d'ajouter des méthodes ou des attributs, ces méthodes et attributs une fois insérés s'afficheront dans le tableau sur la droite.
Il est également possible de générer une interface, la classe implémentant cette interface.

1.1.3.3 Les traitements

Vous pouvez afficher ou non la flèche symbolisant le début d'une séquence.

1.1.3.4 Tous les éléments graphiques

Pour tous les éléments graphiques, la suppression est possible. La suppression d'un élément entrainera la suppression de tous les liens qui s'y raccrochent.

1.2 La programmation

1.2.1 L'architecture

Le démonstrateur est fait de multiples paquetages, nous allons les détailler ici. Le démonstrateur est donc construit entre deux paquetages principaux ihm et evenements.

- evenements
  - btn
    - toolbar
  - menu
    - contextuel
    - fichier
- ihm
  - fenetreClasses
  - fenetreInterdiction
  - menu

ihm est le paquetage ne servant qu'à l'affichage et la création des différentes fenêtres(`FenetreDemo`, `FenetreInterdiction`, `FenetreAjoutAttribut`, ...) mais aussi des menus (`BarreMenus`, `BarreOutilsDessin`, ...). Ce paquetage a ensuite des sous paquetages, classant les fenêtres concernant une classe d'un coté (`FenetreAjoutAttribut` par exemple), les fenêtres d'interdiction lorsque nous ne pouvons créer ou relier deux éléments, et ensuite les différents menus. À la racine du paquetage ihm sont positionnées les classes servant à l'affichage de la fenêtre principale.

evenements quant à lui ne s'axera que sur les évènements qui peuvent avoir lieu dans le démonstrateur, c'est dans ce paquetage que sont créés tous les boutons, et que l'action sur un bouton sera traité.

Dans ce paquetage, nous pouvons donc trouver des sous paquetages pour les boutons, les boutons présents dans la barre d'outils, le menu contextuel, la barre de menus, les événements sur une cellule.

1.2.2 Le graphe et le diagramme

La fenêtre `FenetreDemo` a un objet `panneauGraph`, cette classe hérite de `mxGraphComponent`, cela permet de paramétrer le graphe, la taille, la couleur de fond, afficher une grille ou non, etc...
Cette classe à un `mxGraph`, mais à aussi un `Diagramme`, c'est donc en faisant passer la fenêtre que nous pouvons obtenir toutes les informations nécessaire pour créer un élément.

1.2.3 La création des éléments de modélisation

Chaque élément est créé dans le paquetage evenements.btn.toolbar, dans ce paquetage sont présents tous les événements de la barre d'outils.

1.2.4 La création d'un lien

Les liens eux se font dans la classe `EvenementLienElementGraphique` présente dans le paquetage evenements, un algorithme est présent pour obtenir l'élément source au premier clic, et l'élément de destination au second clic, une fois ces deux éléments récupérés on crée un nouveau Lien.

1.2.5 La suppression d'un élément

La suppression d'un élément peut se faire lorsque le menu contextuel s'affiche, la programmation de l'action `supprimer` se fait dans le paquetage evenements.menu.contextuel, ici sont codées toutes les actions du menu contextuel, et donc supprimer est dans la classe `EvenemenentSupprimer`.

1.2.6 Ajouter des méthodes ou des attributs

Pour ajouter des méthodes ou des attributs, l'affichage de la fenêtre se fait dans le paquetage ihm.fenetreClasses, une fois les champs remplis, l'action sur le bouton valider, l'ajout de la méthode ou de l'attribut se fait dans le paquetage evenements.btn via les classes `BtnValiderAjoutAttribut` et `BtnValiderAjoutMethode`.

Chapitre 2

La bibliothèque UML

La bibliothèque UML est basée sur `JGraphX` cependant, aucune connaissance de cette bibliothèque est indispensable pour se servir de libUML. En effet, son utilisation est volontairement simple, une fois que vous aurez acquis les concepts.

2.1 L'architecture

- eltGraphique
  - *ElementGraphique*
  - eltModelisation
    - *Acteur*
      - ActeurActif
      - ActeurPassif
    - Attribut
    - CasUtilisation
    - Classe
    - *ElementModelisation*
      - Interface
      - Methode
      - Traitement
      - Visibilite
      - Variable
  - ligne
    - Cardinalite
    - Lien
    - MessageTraitement
    - TypeLien
- diagramme
  - Diagramme
  - DiagrammeCasUtilisation
  - DiagrammeClasse
  - DiagrammeSequence

L'architecture est la base d'un projet comme le notre, car il peut être repris et amélioré par d'autres étudiants. En utilisant la notation UML, nous sommes parvenus à élaborer un "méta modèle" de cette notation mais en isolant uniquement l'aspect graphique, faisant ainsi abstraction des multiples règles de conception que la norme UML impose. Après discussion avec le client, de nombreuses modifications ont été apportées à l'architecture d'origine pour aboutir à une solution stable. Cette nouvelle architecture se compose de vingt classes, réparties en quatre paquetages.

Dans cet arbre représentant notre architecture, on peut voir certains noms écrits en gras. ou en *italique*. Les noms en gras représentent les différents paquetages que nous avons séparés.

Ceux en *italiques* sont des classes abstraites crées afin de factoriser le code dans l'optique de réaliser une programmation objet optimale et de suivre les objectifs de propreté du code imposés par le client.

Dans un premier temps, nous avons séparé les diagrammes des éléments graphiques. En effet, un diagramme sera composé de toute sorte d'éléments graphiques. Puis nous avons découpé ces derniers en deux, isolant ainsi les lignes des éléments de modélisation tels que les classes, les traitements ou les cas d'utilisation.

`ElementGraphique` et `ElementModelisation` sont des classes abstraites qui regroupent les fonctions communes à toutes les classes de leur paquetage respectif sans pour autant en fournir une implémentation de chacune d'elles – comme par exemple la méthode qui crée la représentation graphique d'un élément

ou celle qui permet de le supprimer d'un diagramme.
Prenons maintenant chaque paquetage séparément pour voir comment ils fonctionnent plus en détails.

2.1.1 Paquetage eltGraphique

Le paquetage eltGraphique regroupe toutes les classes qui représentent des éléments graphiques. Il regroupe la classe *ElementGraphique* et deux paquetages eltModelisation et ligne. Cette classe possède deux attributs `graph` et `diagramme`, correspondant respectivement au graphe dans lequel sont stockés les éléments et le diagramme affiché à l'écran. Elle comprend également (en plus d'un constructeur initialisant les attributs) les méthodes `supprimer` permettant de supprimer un élément du graphe et du diagramme, ainsi que `creer`, méthode abstraite réimplémentée dans les classes descendantes servant à créer la représentation graphique de l'objet et de l'afficher sur le diagramme.

Paquetage eltModelisation Ce paquetage regroupe toutes les classes représentant les différents éléments de modélisation que l'on trouve dans les diagrammes UML de cas d'utilisation, classes et de séquence. *Acteur* est une classe abstraite car les acteur actifs et passifs ont beaucoup de caractéristiques identiques mais n'ont pas la même représentation sur un diagramme. Les classes `Visibilite`, `Methode` et `Attribut` permettent au client de créer facilement une interface de saisie de ces éléments, facilitant l'usage du logiciel final. De plus, l'ajout de ces méthodes et attributs dans des classes, des acteurs, des traitements ou des interfaces pourra faciliter l'ajout futur de nouvelles fonctionnalités comme par exemple de la génération de code Java.

Paquetage ligne Ce paquetage regroupe peu de classe. `TypeLien` est une classe énumérée servant à recenser tous les types graphiques de liens existant dans la notation UML. `Cardinalite` quant à elle, permet comme `Methode` ou `Attribut`, d'aider le client à permettre une saisie facilité des cardinalités d'un lien dans un diagramme de classe par exemple. La classe `Lien` comprend la méthode `creer` qui va configurer tous les styles de liens et appliquer au nouveau lien le style choisi par l'utilisateur. `MessageTraitement` spécialise Lien, permettant de créer un style de lien particulier aux messages entre traitement dans les diagrammes de séquences.

2.1.2 Paquetage diagramme

La paquetage diagramme comprend plusieurs type de diagramme prédéfinis qui sont cas d'utilisation, classes et séquence. Ils descendent de la classe `Diagramme`. Chaque type de diagramme implémente deux méthodes `eltAutorise` et `lienAutorise` qui permettent respectivement d'autoriser ou interdire un type d'élément particulier et un type de lien entre deux types d'éléments particuliers. La classe Diagramme est générique. Par défaut elle autorise tous les éléments et tous les liens. Il est donc possible au client de réimplémenter ses propres méthodes pour créer ses propres règles.

2.1.3 Diagramme de classes et de paquetages

Si vous souhaitez avoir une idée de la relation entre les classes, le diagramme de classe est disponible annexe C page 15. Également, le diagramme de paquetage, permettant de visualiser également les paquetages est disponible annexe D page 17.

2.2 Le graphe

Le graphe est très important dans la bibliothèque, c'est là où l'élément va s'afficher, ainsi chaque élément graphique doit avoir un `mxGraph`, pour la bibliothèque UML un seul graphe existe et tous les éléments sont dans le même graphe.

2.2.1 mxGraphComponent

`mxGraphCompnent` est un panneau contenant le graphe, cela peut vous permettre de dimensionner votre graphe, d'afficher une grille et d'autres choses intéressantes, pour paramètres votre graphe, nous vous conseillons d'aller voir la documentation de `JGraphX` donnée à l'Avant-propos page ii.

2.2.2 mxGraphControl

La classe `mxGraphControl` (accessible via mxGraphComponent) permet d'ajouter un listener sur les cellules et ainsi pouvoir interagir avec les cellules comme afficher un menu contextuel par exemple. Si vous voulez sélectionner un élément graphique présent sur le graphe, vous devrez utiliser deux méthodes.

- La méthode `getSelectedCell()` qui permet de récupérer une `mxCell`
- La méthode `getElementGraphiqueViaCellule(mxCell)` qui récupère un élément graphique correspondant à la cellule.

2.3 Le diagramme

Le diagramme permet deux choses :

- Il contient tous les éléments graphiques ajoutés dans le graphe, dans un approche de long terme, cela pourrait être utile pour faire une sauvegarde par exemple.
- Il peut permettre de restreindre des actions pour respecter la norme UML. Il vous est possible de redéfinir Diagramme pour refaire vos propres contraintes, mais aussi d'utiliser des contraintes déjà faites pour le diagramme de Classe, diagramme de séquence et diagramme de cas d'utilisation.

2.4 Les éléments de modélisation

2.4.1 Le constructeur

Voici le constructeur d'un élément de modélisation classique :

```
1 ElementModelisation(mxGraph, Diagramme, String, Dimension, Position);
```

- `mxGraph` – Le graphe dans lequel apparaitra l'élément.
- `Diagramme` – Le diagramme dans lequel est l'élément, cela peut aussi servir de contrainte.
- `String` – Le texte qui apparaitra dans l'élément de modélisation (la position de la chaîne de caractère varie en fonction de l'élément de modélisation)
- `Dimension` – La dimension de l'élément
- `Position` – La position de l'élément

2.4.2 Création de l'élément

Pour créer un élément, il suffit d'abord de l'instancier, et ensuite d'appeler la méthode `creer()`.

2.4.3 Suppression d'un élément

Pour supprimer un élément, vous pouvez appeler la méthode `supprimer()` qui supprimera l'élément et tous les liens attachés.
Attention, cela ne supprime que la cellule (`mxCell`), si vous voulez supprimer totalement l'objet, vous devez mettre l'élément graphique à `null`.

2.5 Les liens

La création de lien est ressemble à la création d'un élément de modélisation étant donné que `Lien` et `ElementModelisation` héritent tout deux de `ElementGraphique`.

2.5.1 Constructeur

```
Lien(ElementModelisation, ElementModelisation, mxGraph, Diagramme,
    TypeLien);
```

Les deux premiers paramètres correspondent à la source et à la destination du lien, les deux suivants sont le graphe et le diagramme comme précédemment, le dernier est une énumération des types de liens gérés par la bibliothèque.

2.5.2 Création d'un lien

Il faut instancier le lien, avec la source et la destination. Ensuite, de la même manière que précédemment, il faut faire appel à la méthode `creer()`.
Cependant pour créer des liens entre traitements, il faut utiliser la méthode `ajouterMessage`, qui permet de créer le lien, et d'agrandir le traitement, pour pouvoir mettre d'autres messages ensuite.

2.5.3 Suppression d'un lien

La suppresion d'un lien se faire via la méthode `supprimer()`.
Attention, cela ne supprime que la cellule (`mxCell`), si vous voulez supprimer totalement l'objet, vous devez mettre l'élément graphique à `null`.

2.6 Exemple

```
package com.mxgraph.examples.swing;

import javax.swing.JFrame;
import com.mxgraph.swing.mxGraphComponent;
import com.mxgraph.view.mxGraph;
import eltGraphique.*;
import util.*;

public class Exemple extends JFrame {
  private JPanel panneau;

  public HelloWorld() {
    super("Hello, World!");
    super.add(panneau);
    mxGraph graph = new mxGraph(); // On commence par créer un graphe
        ou apparaîtrons les éléments

    /* on ajoute ca à la fenetre */
    mxGraphComponent graphComponent = new mxGraphcomponent(graph);
    graphComponent.setPrefferedSize(500,500);
    panneau.add(graphComponent);
    Diagramme diagramme = new Diagramme();
```

```
22
23      /* on instancie différents éléments de modélisation */
24      ElementGraphique maClasse1 = new Classe(graph, diagramme, "test 1
           ", new Position(42,42);
25      ElementGraphique maClasse2 = new Classe(graph, diagramme, "test 2
           ", new Position(13,37);
26      Acteur monActeur = new ActeurActif(graph, diagramme, "test 3",
           new Position(13,37);
27
28      /* L'appel à la super méthode créer aura pour effet d'afficher
           nos éléments de modélisation sur le graphe */
29      maClasse1.creer();
30      maClasse2.creer();
31      monActeur.creer();
32
33      /* on instancie des liens en mettant en paramètre la source et la
            destination du lien */
34      Lien monLien1 = new lien(maClasse1, maClasse2, graph, diagramme,
           TypeLien.ASSOCIATION);
35      Lien monLien2 = new lien(monActeur, maClasse1, graph, diagramme,
           TypeLien.SPECIALISATION);
36
37      /* de la même manière, on peut ensuite créer le lien pour l'
           afficher */
38      monLien1.creer();
39      monLien2.creer();
40
41      /* Finalement, on ne voulait pas de l'acteur, alors on le
           supprimer, automatiquement la flèche pointant sur Acteur sera
           supprimée*/
42      monActeur.supprimer();
43      monActeur = null; //on met l'acteur à null pour ne pas perdre de
           mémoire
44    }
45
46    public static void main(String[] args)
47    {
48      Exemple fenetre = new Exemple();
49      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
50      frame.setSize(400, 320);
51      frame.setVisible(true);
52    }
53  }
```

Listing 2.1 – Exemple d'utilisation de libUML

Cet exemple est volontairement simple pour vous permettre de comprendre le fonctionnement de la bibliothèque.

Chapitre 3

Poursuite de développement de la bibliothèque

3.1 JGraphX

La bibliothèque utilise entièrement `JGraphx` pour dessiner les composants. Nous avons fait en sorte qu'un utilisateur voulant se servir de libUML n'ai nullement besoin de la connaissance de `JGraphx`, ainsi, il faut restreindre l'utilisation de `JGraphx` au strict minimum (`mxGraph`...) quitte à redéfinir des fonctions appelant la fonction parente, ça a l'avantage d'intégrer la Javadoc de cette méthode à la documentation du projet.

La bibliothèque `JGraphX` a le grand défaut de n'avoir qu'une toute petite communauté, ainsi, vous trouverez rarement des réponses à votre problème sur un forum, nous avons donc beaucoup utilisé la documentation (Javadoc et manuel), le lien de cette documentation est disponible dans l'Avant-propos page ii.

Les développeurs de `JGraphX` ont effectué un exemple de leurs bibliothèque, celui-ci est extrêmement complet et puissant, de la même manière que le but de notre démonstrateur est de montrer les possibilités de la bibliothèque, leur exemple effectue toutes les possibilités de `JGraphX`, nous nous en sommes aidés, ainsi nous avons eu des exemples. Celui-ci est accessible sur notre serveur
▷ http://telechargements.joohoo.fr/JGraphXExamples.zip

3.2 Création d'un nouvel élément

Pour créer un nouvel élément, il faut créer une classe héritant de `ElementGraphique` ou pour les liens de `ElementModelisation`, redéfinir la méthode `creer`, dans cette méthode, le style doit être créé, c'est dans ce style que l'ont peut choisir la forme, la couleur, la possibilité de le déplacer etc... Comme ceci par exemple :

```
1 private void creerStyleActeurActif() {
2   Map<String, Object> nouveauStyle = new HashMap<String, Object>();
3   mxStylesheet feuilleStyles = super.getGraph().getStylesheet();
4   nouveauStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_ACTOR);
5   /* opacité */
6   nouveauStyle.put(mxConstants.STYLE_OPACITY, Constantes.OPACITE);
7   /* couleur du texte */
8   nouveauStyle.put(mxConstants.STYLE_FONTCOLOR, Constantes.
      COULEUR_TEXTE);
```

```
9   nouveauStyle.put(mxConstants.STYLE_STROKECOLOR, Constantes.
        COULEUR_BORDURE); //Couleur de la bordure
10  /* Un élément peut s'inclure dans l'acteur*/
11  nouveauStyle.put(mxConstants.STYLE_FOLDABLE, mxConstants.NONE);
12
13  /*Création du style avec les paramètres précédent */
14  feuilleStyles.putCellStyle("ACTEUR_ACTIF", nouveauStyle);
15 }
```

Listing 3.1 – Création du style de l'acteur actif

Pour créer un nouveau style de flèche, le principe est le même, il faut créer un style, sauf qu'il faut ajouter ce style de flèche dans l'énumération, dans le `switch` présent dans la classe `Lien`.
Cependant, nous n'avons pas trouvé comment changer la couleur de fond du bout de la flèche.

Chapitre 4

Conclusion du projet

Le projet de départ étant une étude de faisabilité, nous pouvons donc en conclure que c'était tout à fait faisable, et que ce n'était pas très compliqué de pouvoir tracer des éléments de modélisation, nous avons cependant eu beaucoup de mal à trouver une architecture qui soit viable, ceci venant peut être aussi de notre manque d'expérience. L'architecture a mis longtemps à être posée, mais je pense que maintenant elle semble stable.

`JGraphX` était un bon choix de bibliothèque, ses possibilités sont assez grandes, grâce à cette bibliothèque nous avons put faire une bibliothèque assez simple à continuer , l'architecture ayant été mise en place, et le principe de développement créer, il suffit de créer de nouveaux styles comme expliqué section 3.2 page 10.

4.1 Un outil UML complet ?

Après réflexion, nous pensons que développer un outil UML complet nous prendrait environ **6 mois**, en utilisant ce que nous avons déjà développé de notre bibliothèque. `JGraphX` nous permet de faire certaines choses vraiment simplement, pour certaines autres choses, cela est beaucoup plus difficile, mais avec du temps, nous pensons que c'est faisable.

4.2 Utilité d'un outil fragmenté ?

Étant donné que nous sommes étudiants, et ne travaillons que sur des petits projets, nous n'avions pas tout de suite vu l'utilité d'un outil fragmenté, cependant en effet, cela est utile pour de gros projets.

En effet, certaines personnes sont "pollués" avec beaucoup de fonctionnalités dont ils n'ont pas besoin, cela leurs permettrai de se concentrer uniquement sur leur projet.

Également, cela pourrait créer un système de droits, et ainsi éviter qu'un diagramme soit modifié sans autorisation. Notre bibliothèque UML permettrai de réaliser ceci, grâce aux contraintes que nous avons mis en œuvre.

Annexe A

Glossaire

Bibliothèque (p i) – Composant programmé dans un langage donné fournissant des méthodes permettant d'effectuer des tâches voulus

Java (p i) – Langage de programmation orienté objet moderne, il compile le programme pour ensuite l'exécuter sur une machine Java, ainsi le programme une fois compilé peut être exécuté sur différentes plateformes (Windows, Linux, Mac OS X, ...).

Javadoc (p 1) – Javadoc est un système de documentation automatique pour Java, cette documentation est générée au format HTML. Il analyse les commentaires introduits dans le programme, ceux-ci étant un peu particuliers.

Paquetage (p 5) – Un paquetage en Java est un regroupement de classes ayant la même thématique.

UML (p i) – (Unified Modeling Language) Langage de modélisation graphique à base de pictogramme. Il est apparu dans le monde du génie logiciel dans le cadre de la conception orientée objet. Ce langage est composé de différents diagrammes, allant du développement à la simple analyse des besoins.

Annexe B

Liste des liens

B.1 Documentation

Tous les liens se rapportant à la documentation de notre bibliotheque ou de la bibliotheque `JgraphX`.

▷ http://documentation.joohoo.fr/libUML/bibliothequePrivee/index.html
▷ http://documentation.joohoo.fr/libUML/bibliothequePublique/index.html
▷ http://documentation.joohoo.fr/libUML/demonstrateurPrivee/index.html
▷ http://documentation.joohoo.fr/libUML/testsUnitaires/index.html
▷ http://documentation.joohoo.fr/JGraphX/index.html

L'adresse suivante récapitule les liens pour la documentation de notre bibliothèque :

▷ http://documentation.joohoo.fr/libUML/

B.2 Téléchargements

Tous les liens de téléchargement, permettant de télécharger des bibliothèque, ou des exemples.

▷ http://telechargements.joohoo.fr/libUML/libUML-netbeans.zip
▷ http://telechargements.joohoo.fr/libUML/JGraphX-1.8.jar
▷ http://telechargements.joohoo.fr/libUML/JGraphXExamples.zip

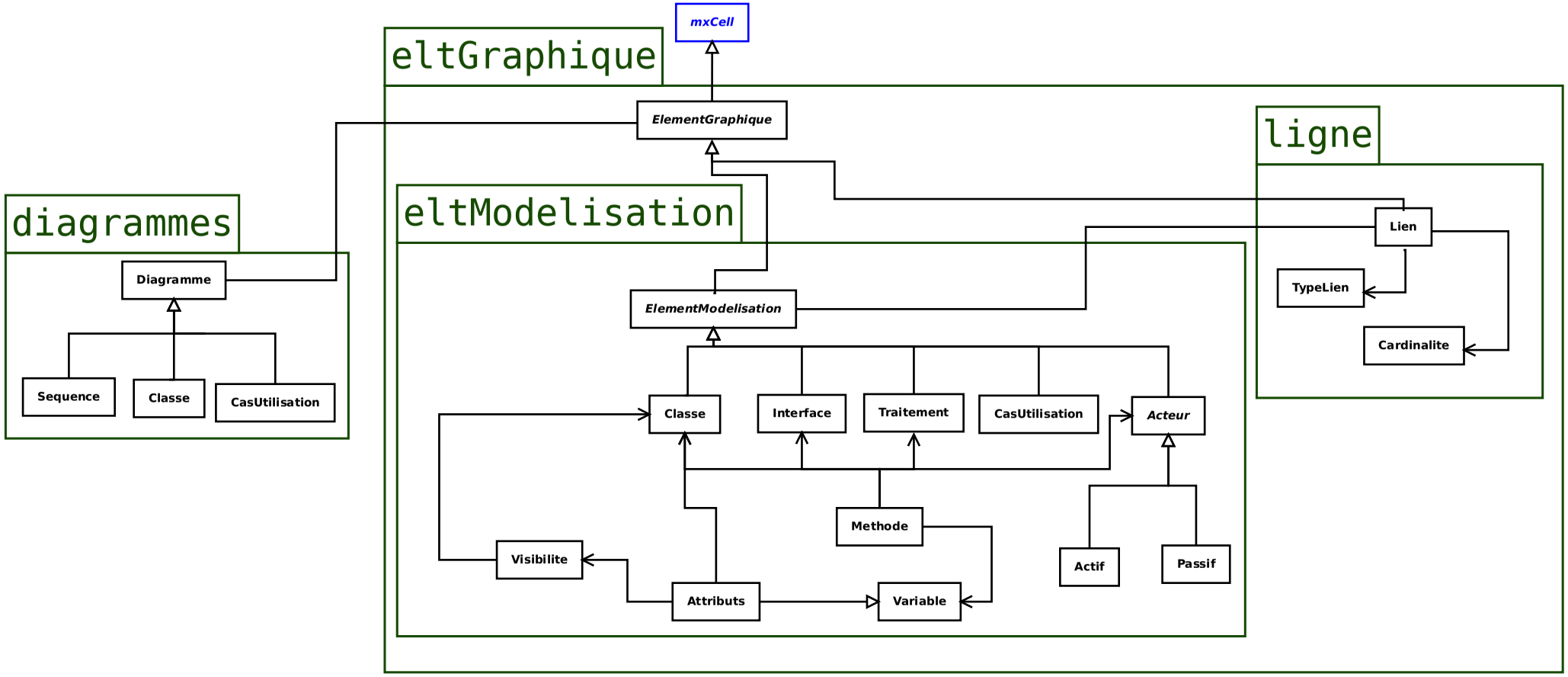
L'adresse suivante récapitule les liens pour les téléchargements en rapports avec libUML.

▷ http://telechargements.joohoo.fr/libUML/

Annexe C

Diagramme de classes

Bibliothèque UML



Annexe D

Diagramme de paquetages

Bibliothèque UML