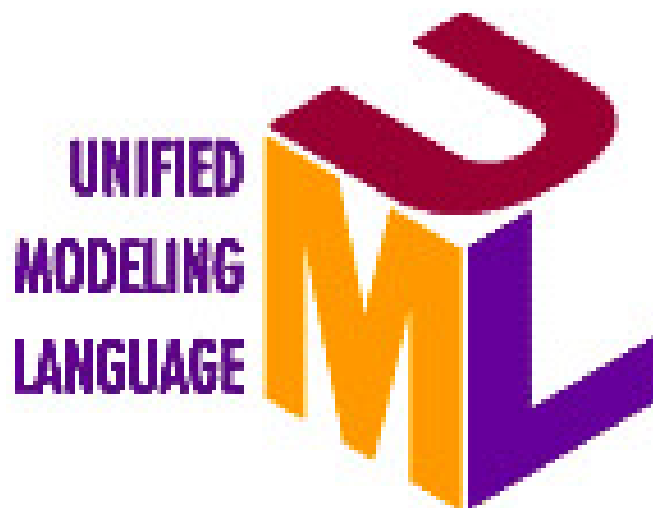


GUYARD Matthieu
LALLEMAND Michael
MEUSBURGER Matthias
POTHIER Damien
TRUCHE Damien

GENERATEUR DE MODELES UML



SOMMAIRE

<u>INTRODUCTION</u>	<u>2</u>
<u>I - PRESENTATION DU SUJET</u>	<u>3</u>
<u>II - PRESENTATION DES FONCTIONNALITES DE L'APPLICATION</u>	<u>6</u>
<u>III – LIMITES ET EVOLUTION POSSIBLES DE L'APPLICATION</u>	<u>15</u>
<u>IV – ASPECT HUMAIN</u>	<u>18</u>
<u>CONCLUSION</u>	<u>21</u>

INTRODUCTION

Développée en 1994 afin de réaliser l'unification des principales méthodes objets, Booch et OMT, la méthode UML (Unified Modeling Language) constitue aujourd'hui l'inévitable travail préparatoire à toute conception de programme orienté objet. Cependant, effectuer cette tâche à la main peut se révéler rapidement fastidieux.

C'est pourquoi nous avons choisi pour notre projet tutoré de deuxième année de concevoir un programme permettant l'élaboration de modèles UML.

Nous allons ici essayer de vous présenter les principales caractéristiques de notre application, bien qu'il soit nécessaire d'explicitier en premier lieu les notions de base qui constituent la modélisation UML. Nous allons également rendre compte des limites de notre logiciel ainsi que de ses évolutions possibles. Finalement, nous allons parler de l'aspect humain, composante intrinsèque à l'élaboration de ce projet.

I) Présentation du sujet

I.1) La modélisation U.M.L :



L'UML (Unified Modeling Language) est une méthode permettant de représenter sous forme de schémas les différents composants d'un programme orienté objet, à la manière de la méthode MERISE qui permet de représenter la structure d'un système d'information. Les différentes notions manipulées dans les modèles U.M.L sont :

- **Les classes** (qui peuvent être considérées comme équivalentes aux entités de MERISE)
Une classe peut être concrète ou abstraite. Elle est abstraite si aucun objet de cette classe n'est instancié. Si une classe est abstraite, son nom est noté entre accolades
→ { nomClasse }
- **Les attributs** (qui peuvent être considérés comme équivalents aux propriétés de MERISE)
Un attribut possède un type et peut être d'accès public (+), privé (-) ou protégé (#).
- **Les méthodes**
Une méthode possède un type de retour (éventuellement nul) et une suite (éventuellement nulle) de paramètres. A la manière des attributs, les méthodes ont un accès qui peut être, soit public, soit privé, soit protégé.

étudiant
- <i>Chaine</i> nom - <i>Entier</i> numEtudiant
+ <i>etudiant</i> (<i>Chaine</i> nom, <i>Entier</i> numEtudiant) + <i>Chaine</i> RenvoieNom() + <i>Entier</i> RenvoieNum()

- **Les liens**, qui servent à représenter les relations entre les différentes classes.

Ils peuvent être de 4 types :

- Association → Représente deux classes conceptuellement au même niveau.



- Héritage → La classe fille hérite des propriétés de la classe mère.



- Agrégation → Représente la notion « partie/ensemble » entre deux classes.



- Composition → Comme l'agrégation, la composition représente la notion « partie/ensemble » entre deux classes, à la différence qu'ici, la vie de la classe « partie » est liée à celle de la classe « ensemble »



On notera par ailleurs dans ces liens (excepté l'héritage) la présence du concept de cardinalité, qui encore une fois, se retrouve dans la méthode Merise.

I.2) Pourquoi un générateur de modèles UML ?

Tout d'abord, il n'existe à notre connaissance aucun logiciel de modélisation UML dans le domaine public. Nous pensions ainsi pouvoir mettre à disposition les fonctionnalités de base d'un logiciel de ce type, qui puisse, pourquoi pas, être repris et amélioré par d'autres.

I.3) Choix de conception :

Nous avons adopté pour la programmation de ce logiciel, le langage JAVA, qui est lui-même un langage objet, et nous semblait donc tout particulièrement adapté à ce sujet. En ce qui concerne l'interface graphique, nous avons choisi d'utiliser les composants Swing, notamment pour les plus grandes possibilités offertes par rapport à Awt. Finalement, le système d'exploitation utilisé a été choisi librement par chaque membre de groupe. Le java étant portable, cela n'a pas posé de problème majeur lors de la phase de développement.

II) Présentation des fonctionnalités de l'application

L'éditeur de modèle UML a été conçu pour une utilisation privée et est caractérisé par une économie des fonctions, des contrôles simples et une interface graphique accueillante qui permettent une prise en main rapide pour des débutants en UML.

Cette application n'est ni de très haut niveau, ni très riche, c'est pourquoi ses menus simples mais efficaces apportent à l'utilisateur un réconfort permettant de mettre en forme un modèle UML.

L'application met en œuvre les principaux concepts de la modélisation comme la notion de classe, de lien, d'attribut ou encore de méthode. Les notions complexes de la modélisation étant délaissées pour simplifier l'application. Nous verrons donc dans un premier temps les différentes fonctionnalités de ce petit logiciel agréable et fonctionnel.

II.1 Création des différents objets

II.1.1 Création d'une classe

Rappelons tout d'abord ce qu'est la notion de classe : la classe est l'élément central de la modélisation UML comme pour le langage JAVA. Celle-ci est composée d'un nom, d'un type, de ses attributs et enfin de ses méthodes. Le tout compose la classe qui sera à la base de notre application.

Le nom d'une classe ne sera, premièrement, pas unique pour une classe c'est à dire que plusieurs classes pourront porter le même nom mais chaque classe sera caractérisée par sa liste d'attributs et de méthodes et deuxièmement ce nom sera accompagné de son type représenté traditionnellement comme le veut la norme par des accolades entourant le nom de la classe dans le cas d'une classe abstraite et sans accolade dans le cas d'une classe concrète.

A la création d'une classe, son nom ne sera aucunement vérifié mais admettra uniquement des caractères alphanumériques.

Exemple :



En ce qui concerne son type, l'utilisateur sera limité dans son choix par une liste déroulante indiquant les seules possibilités autorisées.

Illustration :



Pour ce qui est de la forme générale, chaque classe sera dessinée en trois parties distinctes :

- l'entête de la classe sera occupée par son nom et son type,
- la partie centrale définira ses attributs,
- enfin le bas de page énumèrera ses différentes méthodes.

Schéma type d'une classe telle qu'elle est affichée par notre application :

toto
+ int at
Public int methodetoto (double para2 ,float para1)

On considèrera une classe achevée lorsque son type et son nom seront entrés par l'utilisateur.

II.1.2 Création d'un attribut

Les attributs d'une classe recensent toutes les variables qui vont être utilisées par cette classe au niveau local. C'est pourquoi il est nécessaire de définir un type, un nom, et un accès.

L'adaptation de la taille de la classe en fonction de ses données étant décrite plus loin, nous allons étudier la procédure de création d'un attribut.

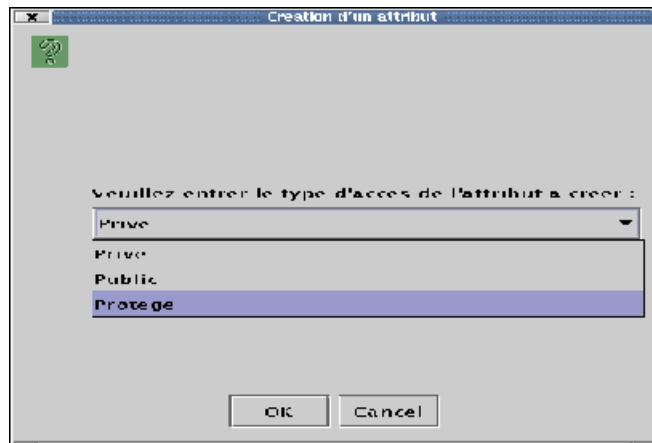
Au début de la création d'un attribut, sa classe propriétaire n'est pas connue et chaque information est saisie au fur et à mesure par l'utilisateur.

On débute donc avec le nom de l'attribut saisi de la même manière que celui de la classe.

Ensuite, on doit connaître le type de celui-ci. Le type d'un attribut représente ce que la variable va contenir comme nature de donnée. Ici, le type pouvant être de diverses natures, on laisse l'utilisateur le rentrer.

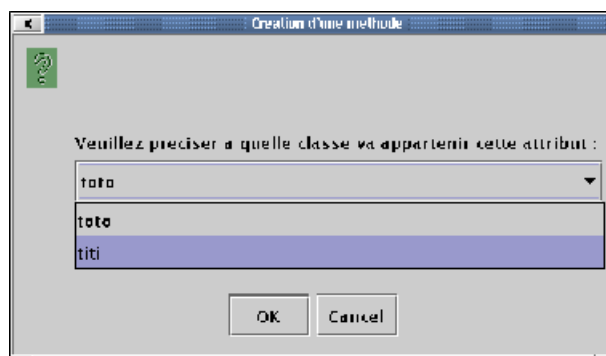
Mais un attribut possède également un type d'accès. Cet accès correspond au comportement de la variable vis-à-vis d'autres classes. Pour respecter la norme de la modélisation UML, rappelons que nous l'avons représenté sous forme de trois symboles : +, -, # respectivement pour un accès public, privé et protégé. Le choix étant limité à ces trois types d'accès, il est demandé par une liste déroulante indiquant les seuls choix autorisés.

Exemple :



Ensuite il est attribué à la classe choisie par l'utilisateur grâce à une liste des classes susceptibles de le recevoir ce qui permet à l'utilisateur de ne pas se faire d'erreur quant au propriétaire.

Illustration :



Nous avons vu comment créer un attribut et l'affecter à une classe, mais une classe est également susceptible de comporter des méthodes. C'est ce que nous allons étudier maintenant.

II.1.3 Création d'une méthode

Une méthode peut être vue comme une fonction, elle accepte donc des paramètres, ce qui va nous amener à découper notre étude en deux sous-parties.

Tout d'abord, tout comme la création d'un attribut, son nom, et sa classe propriétaire sont rentrés. Mais une méthode possède en plus un type de retour. Comme pour le type d'un attribut, le type de retour d'une méthode sera saisi par l'utilisateur.

Ensuite, chaque paramètre est rentré un par un. Un paramètre possède un type et un nom qui sont demandés de la même manière qu'un attribut pour accentuer le caractère agréable de la saisie.

De la même façon que pour un attribut, chaque méthode est créée indépendamment de la classe qui en est la propriétaire. C'est une fois que la méthode est définie que l'utilisateur va l'attribuer à une classe via un menu déroulant.

Exemple :

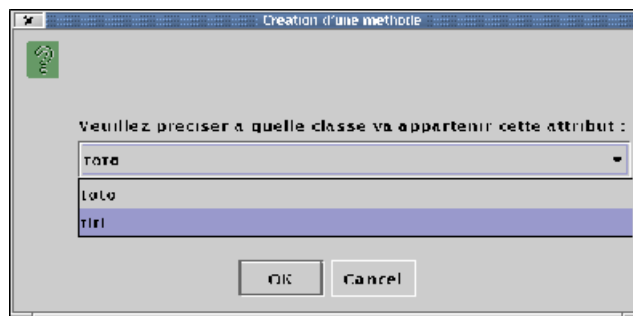


Illustration :

Méthode →

toto
- int at2
float at3
- double at
+ int methode1 (int parametre2 ,double parametre1)

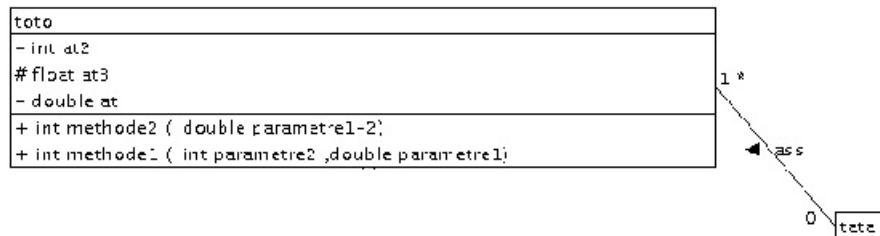
II.1.4 Création de liens

Nous arrivons à l'élément le plus complexe de l'application et le plus difficile de mise en œuvre tant par la diversité que par le souci de clarté.

C'est ainsi que nous avons décidé de ne pas représenter certains liens comme la dépendance qui n'est pas indispensable. Les autres types de liens comme l'association, la composition, l'agrégation ou encore l'héritage sont disponibles et sont représentés selon la normalisation UML. La notion de lien apporte également celle de sens car un lien est toujours orienté. Ce qui nous amène à étudier de plus près ces différents liens.

Tout d'abord, la relation d'association sera modélisée par une simple droite entre deux classes surmontée par son nom ainsi qu'un triangle indiquant le sens de celle-ci.

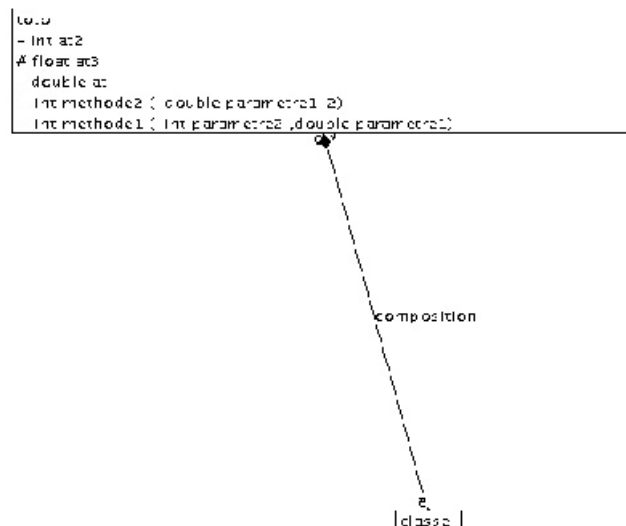
Illustration :



Quant à la composition et à l'agrégation, elles seront respectivement représentées par un trait se terminant par un losange plein et un trait se terminant par un losange vide.

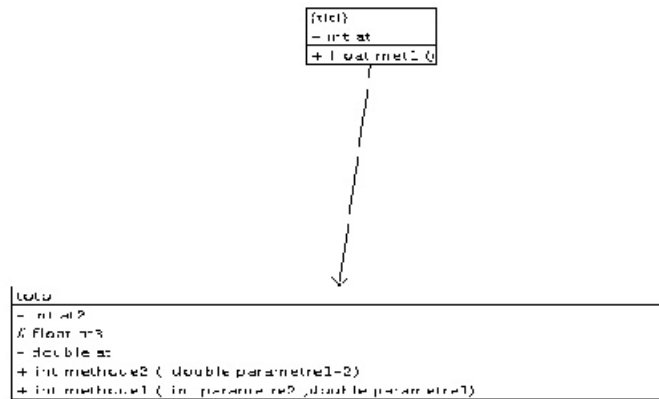
Illustrations :

- Compositions :



Dans le cas de l'héritage une simple flèche orientée vers la classe mère sera dessinée.

Illustration :

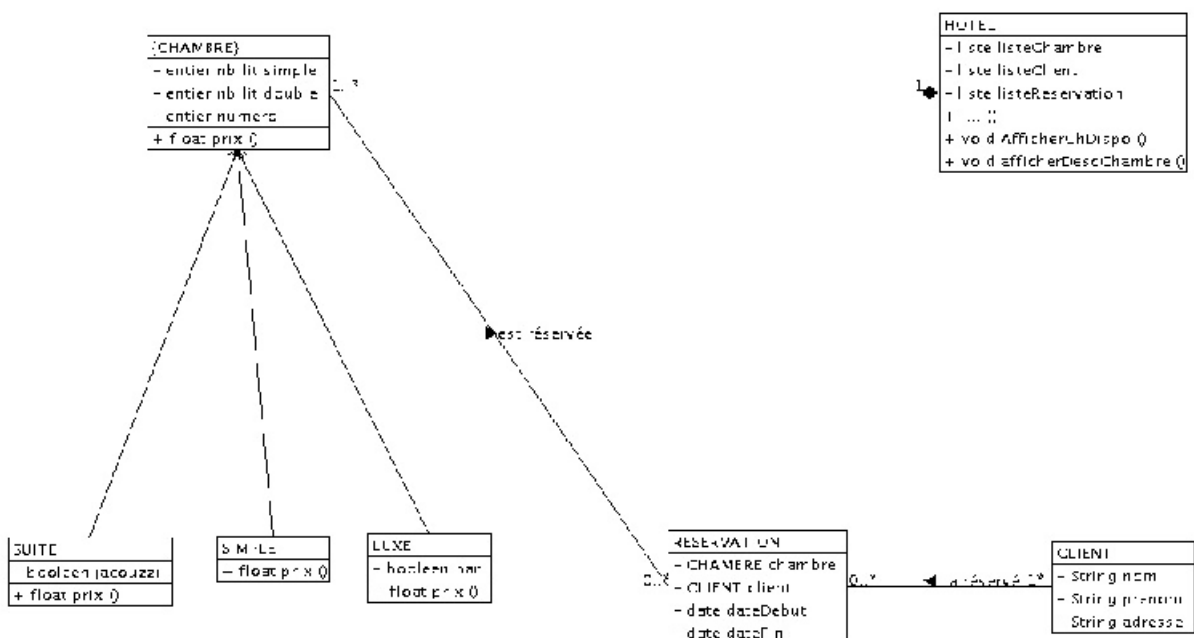


Dans les trois premiers cas, les cardinalités seront éditées sur le lien et se déplaceront avec ceux-ci lors de la mise en forme manuelle. L'entrée des cardinalités ne faisant l'objet d'aucun test, l'utilisateur pourra afficher les cardinalités qu'il souhaite. Le lien héritage ne faisant état d'aucune cardinalités, celles-ci ne sont pas sujettes à une saisie utilisateur.

Illustration d'un modèle UML complet :

Sujet : gestion d'un hôtel

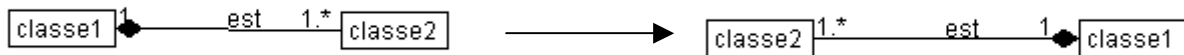
Nous allons voir un modèle UML complet c'est à dire avec des classes comportant des attributs et des méthodes mais aussi différents liens.



II.2 Déplacement

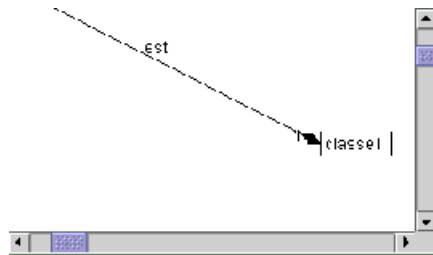
Pour pouvoir organiser son modèle de manière claire et lisible, l'utilisateur aura la possibilité de déplacer les différentes classes créées en cliquant sur la classe intéressée et en déplaçant la souris. Bien entendu, les différents liens que possède la classe à déplacer s'orienteront automatiquement en fonction des mouvements de la classe.

Déplacement d'une classe :



L'application permettra également de gérer un modèle d'une taille assez conséquente puisque le panneau graphique utilisable dépasse la taille de la fenêtre et l'utilisateur pourra se mouvoir dans la zone de travail à l'aide de barres de défilement présentes dans le bas et à droite de la fenêtre.

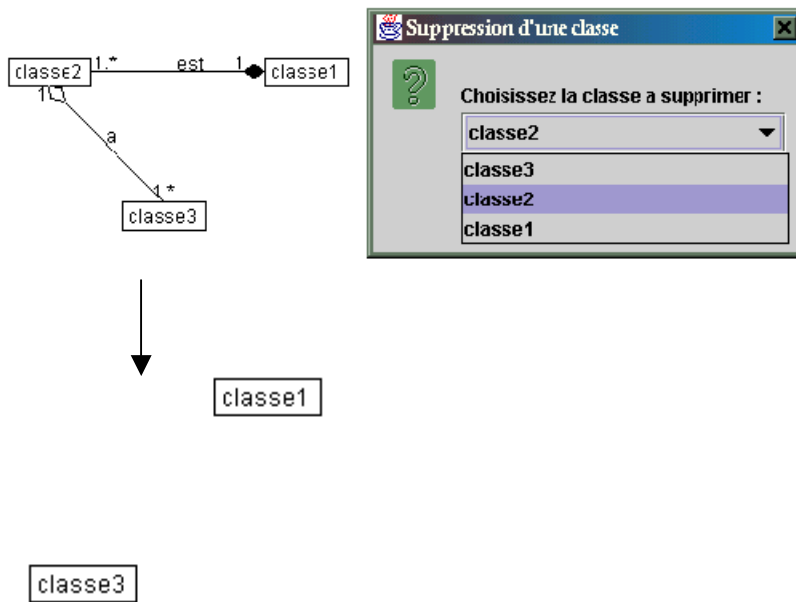
Déplacement en utilisant les barres de défilement :



II.3 Suppression

L'utilisateur pourra choisir de supprimer une certaine classe ou un certain lien tout en sachant que lorsqu'il supprimera une classe, tous les liens se rattachant à cette classe seront supprimés.

Exemple : suppression d'une classe :

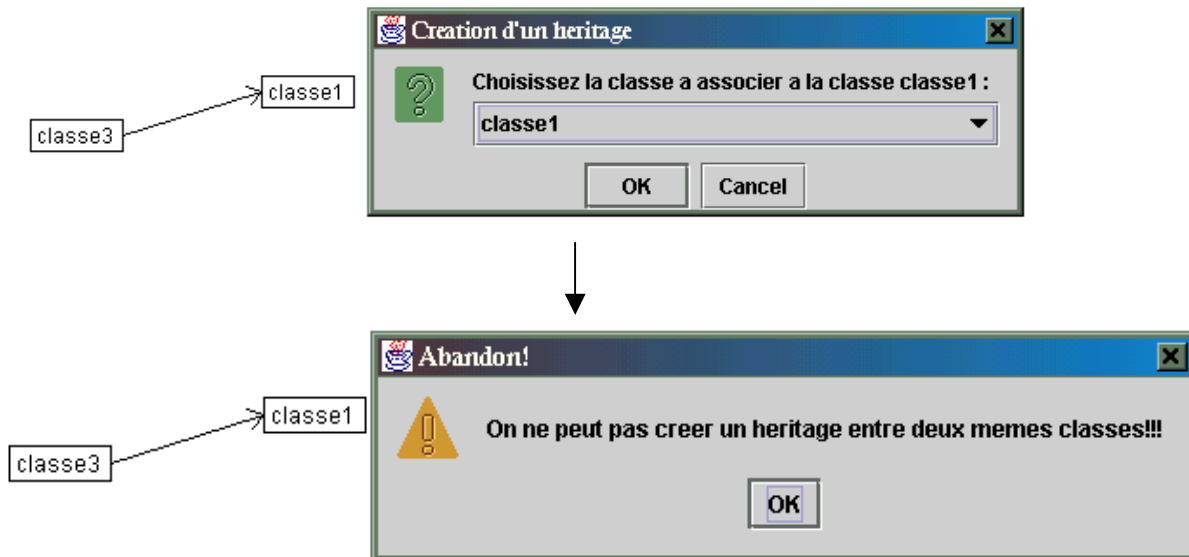


Pour ce faire, suivant qu'un lien ou une classe doit être supprimé(e), une boîte de dialogue apparaîtra contenant un menu déroulant par le biais duquel l'utilisateur pourra choisir la classe ou le lien à supprimer. Cependant, en ce qui concerne la suppression d'un lien, celui-ci est identifié par la sélection des deux classes qu'il relie. Lorsqu'une classe ou un lien est retiré(e), elle ou il n'apparaît évidemment plus dans le menu déroulant.

II.4 Protection

Nous avons prévu les cas où l'utilisateur souhaiterait effectuer des actions impossibles ou dénuées de sens. Par exemple, s'il veut relier une même classe par un lien, le programme le préviendra par une boîte de dialogue de l'impossibilité d'accomplir cette action.

Exemple: interdiction de créer un lien entre une même classe :



Par contre, l'utilisateur a une entière liberté sur le choix des cardinalités. En ce qui concerne les méthodes et les attributs, le programme ne l'empêchera pas de créer un champs (type, nom, retour) vide. Lorsque l'utilisateur décide de créer un nouveau modèle (cliquant sur nouveau), un message apparaîtra lui demandant s'il veut vraiment quitter le modèle actuellement en cours d'utilisation.

II.5 Sauvegarde / Restauration

L'application donne la possibilité de sauvegarder le modèle en passant par le menu « fichier » puis l'option « enregistrer » ou « enregistrer sous ». Si le modèle a été enregistré précédemment, le fait de cliquer sur « enregistrer » enregistrera le programme sous le même nom de fichier alors que l'action « enregistrer sous » propose une boîte de dialogue dans laquelle l'utilisateur entre le nom de fichier voulu. En ce qui concerne la restauration, si l'utilisateur souhaite charger un fichier préalablement sauvegardé, il pourra cliquer sur « charger » et entrer le nom du fichier qu'il souhaite restaurer. Le fichier créé par la sauvegarde n'est pas lisible avec un éditeur de texte, il sert seulement à la restauration, et s'il est modifié, la restauration ne pourra pas être faite.

III) Limites et évolutions possible de l'application

Dans cette partie, les limites ainsi que les évolutions possibles de notre application seront présentées. On pourrait se demander pourquoi différencier ces deux catégories puisque chaque limite de l'application peut être également considérée comme une évolution possible. En fait, les limites de l'application sont constituées de défauts présents dans notre version de l'application, alors que les évolutions possibles n'ont pas encore eu d'implantation dans notre programme.

III.1 Limites de l'application

- En ce qui concerne la sauvegarde et la restauration d'un modèle UML :
 - L'absence de fenêtre de balayage de l'arborescence et de visualisation des fichiers implique que :
 - les fichiers sont sauvés et restaurés dans le répertoire courant
 - l'utilisateur doit se souvenir du nom de fichier du modeler à restaurer.
 - L'absence de test concernant l'écrasement d'un fichier existant implique que l'utilisateur doit faire attention au nom de fichier donné lors de la sauvegarde.
- Il n'existe pas de possibilité de modifier un objet du modèle. En effet, pour modifier un objet, l'utilisateur doit supprimer cet objet et le recréer. Cette contrainte, qui peut s'avérer mineure lorsqu'il s'agit de modifier l'attribut d'une classe peut l'être beaucoup moins lorsqu'il s'agit de changer le nom ou l'accès d'une classe, puisque la suppression d'une classe entraîne automatique la suppression de tous les liens qu'elle possède.
- La suppression des méthodes n'a pas été implémentée
- S'il est possible, comme indiqué dans la partie 2, de déplacer une classe avec la souris, cette technique introduit une faiblesse. En effet, le déplacement d'une classe s'effectuant par le coin haut/gauche de celle-ci, si l'utilisateur la déplace à l'extrémité basse ou à l'extrémité droite du panneau graphique, et la laisse à cet endroit, c'est-à-dire à l'extérieur du panneau, il lui sera impossible de la récupérer. Cette faiblesse est tout de même palliée par le fait que le panneau graphique est beaucoup plus grand que la fenêtre elle-même puisque ses dimensions sont de 3000 pixels par 3000. Ainsi, les chances qu'un modèle s'étende jusqu'à un de ces bords sont faibles.

III.2 Evolutions possibles

L'éditeur UML présenté est opérationnel. En effet il gère toutes les fonctions nécessaires à l'élaboration d'un modèle. Cependant, il présente quelques lacunes notamment au niveau des données, au niveau graphique, mais aussi au niveau de la simplicité d'utilisation.

Une meilleure gestion des données :

Le programme présenté permet de créer et de supprimer les données à l'aide de deux menus. Pour modifier une donnée il faut donc la supprimer et la recréer. Une évolution possible est la création d'un menu similaire aux deux précédents qui permettrait de modifier toutes les données de la même manière qu'une suppression. Il serait possible de modifier ainsi chaque partie du modèle.

Cette évolution paraît tout à fait réalisable : au niveau de la programmation des différents objets UML, chaque donnée possède des accesseurs en modification.

De plus, le filtrage des données aurait pu être mieux géré. Par exemple, en ce qui concerne les cardinalités l'utilisateur est libre de rentrer n'importe quelle chaîne de caractère, ce qui pose des problèmes à l'affichage si cette chaîne est trop longue.

III.2.1 Les améliorations graphiques

L'application UML affiche à l'écran tous les objets créés de la même couleur. Une évolution simple consiste à gérer les couleurs : dessiner chaque objet du même type de la même couleur (ex : toutes les classes en rouge).

D'un point de vue pratique, il est possible d'ajouter une barre d'outils avec des raccourcis pour permettre à l'utilisateur d'accéder directement à certaine partie du logiciel sans être obligé d'accéder à chaque fois au menu.

Une autre évolution est la gestion de la sélection des objets à l'écran. Ainsi lorsque l'utilisateur clique sur une classe, cette dernière changerait de couleur. L'évolution permettrait de réduire le temps de saisie de l'utilisateur. Par exemple, lorsque l'utilisateur voudrait créer un lien, il cliquerait sur l'icône « création de lien » et ensuite sur les 2 classes entre lesquelles se trouve le lien.

De plus, il faudrait aussi gérer l'agrandissement ou la réduction des classes. En effet, dans le projet les classes prennent la taille de la plus longue ligne. Ceci donnerait plus de liberté à l'utilisateur pour agencer son modèle.

De même la possibilité de déplacer les liens apporterait plus de lisibilité à l'application. En effet l'affichage d'un lien se fait entre les milieux de chaque classe. Un problème survient donc lorsque qu'une classe est reliée à 2 autres qui se trouvent du même côté. Les cardinalités des deux liens se superposent.

Une autre évolution graphique consisterait à gérer le clic droit pour faire apparaître un menu contextuel. On afficherait ainsi les opérations relatives à l'élément sur lequel on a cliqué.

La dernière évolution graphique serait la possibilité de modifier directement sur le dessin les propriétés de chaque propriété d'un objet. On raccourcirait considérablement la saisie des données et on augmenterait la simplicité d'utilisation.

III.2.2 Impression du modèle

La fonction principale d'un éditeur UML est de représenter un modèle plus propre et plus clair qu'un dessin réalisé à la main. L'impression est donc une évolution nécessaire à l'utilisation future du projet. En effet, l'utilisateur ne trouvera pas le logiciel d'un grand intérêt s'il ne peut pas par la suite imprimer le modèle qu'il a réalisé. De plus, l'implémentation d'un gestionnaire d'impression en java n'est pas extrêmement difficile à mettre en place.

III.2.3 La gestion de plusieurs modèles en même temps

Une autre évolution consisterait à gérer plusieurs modèles à la fois dans le but de pouvoir exporter les éléments de l'un vers l'autre. Ceci implique la création d'un menu édition avec lequel il serait possible de dupliquer des classes par exemple (copier coller). En pratique, au niveau des données, il suffirait d'ajouter une liste de modèle au sein du programme. Mais cette évolution n'a de sens que si l'utilisateur peut passer d'un modèle à l'autre à tout moment, via un menu affichant les différentes fenêtres ouvertes.

De nombreuses évolutions sont donc possibles pour ce projet. Elles sont plus ou moins intéressantes et demandent un tant soit peu de travail. Avec un peu plus de temps, il aurait été possible de s'occuper de la suppression des méthodes, de la modification des données ou de la gestion des couleurs sans grande difficulté. Quant aux différentes améliorations graphiques ou à la gestion « multi-modèles », elles demanderaient sans doute des études plus approfondies.

IV) L'aspect humain

Au long de cette partie, nous aborderons l'aspect humain de notre projet, c'est à dire tout ce qui concerne la gestion du planning, la gestion des différentes versions de chacun et la répartition du travail au sein du groupe. Tous ces points ont été, à un degré moindre, des problèmes à résoudre. Nous verrons les choix pris en terme d'organisation pour palier à ces problèmes.

IV.1 Problèmes de planning

Le respect du planning et des objectifs n'a pas été aisé, en effet au départ du projet, le groupe destiné à le réaliser était composé de six personnes mais un de nous ayant quitté l'IUT et par la même occasion le projet, l'équipe se vu donc réduite à cinq personnes.

Au départ, nous avons divisé le groupe en trois sous-groupes de deux programmeurs pour avoir une meilleure productivité. Lorsque deux personnes travaillent ensemble, le travail se fait plus vite et mieux et il y a moins de conflits qu'à trois ou plus. Cette organisation a du être remise en cause avec le départ de notre collègue.

Il a donc fallu réorganiser le travail au sein du groupe pour qu'une personne ne fasse pas le travail attribué au départ à deux personnes. Il a également été difficile de reprendre le travail fait par la personne qui a quitté le groupe et de l'intégrer à l'application. Nous avons ainsi pris un retard non négligeable sur le planning.

Ceci va nous amener à parler des problèmes de rassemblement des différentes versions du programme faites ou complétées par chacun.

IV.2 Problèmes de versions

A certains moments de la conception de l'application, il était indispensable de mettre en commun les versions de chacun. Il fallait donc que chaque personne sache parfaitement dans quelles parties du programme elle avait ajouté ou modifié du code, nous nous sommes donc fiés à la mémoire de chacun ou aux notes prises par le programmeur lorsqu'il a ajouté ou modifié des lignes de codes ou des fichiers du programme.

Ce projet étant notre premier projet, nous n'avions pas prévu ce problème, mais pour nos prochains projets, nous pourrions adopter l'utilisation d'un outil permettant de gérer les différentes versions d'une application comme le fait CVS vu cette année dans le cours de Système.

IV.3 Répartition du travail

Voyons maintenant de quelle façon les tâches ont été réparties au sein du groupe. Matthias s'est occupé de concevoir l'interface graphique, en résumé tout le système de menu destiné à l'utilisateur, ainsi que du déplacement des classes avec la souris. Il a aussi géré toutes les versions du programme, c'est lui qui mettait en commun les parties de chacun.

Damien T s'est consacré au système de sauvegarde et de restauration des modèles créés à l'aide de l'application.

Michaël, Damien P et Matthieu se sont occupé du dessin du modèle à l'écran (Classes, liens...) et de la représentation des données dans le programme c'est-à-dire représenter en JAVA un modèle UML.

Cependant, il serait restrictif de s'arrêter à cette description du travail, puisque chacun s'est intéressé aux parties des autres, et qu'au final, même si effectivement la programmation s'est déroulée en grande partie selon cette répartition, nous avons tous une bonne connaissance globale du programme.

On peut surtout noter que l'ambiance était bonne au sein du groupe, avec une entraide collective. Par exemple quand un de nous avait un problème concernant la partie qu'il avait à réaliser, il pouvait sans problèmes demander conseil aux autres membres du groupe qui essayaient de lui donner une solution. En résumé, chacun de nous était au service des autres, il n'y avait pas de rivalité au sein du groupe.

Il est clair que tous les choix pris pour l'organisation du travail ne sont pas les meilleurs, mais cela nous a montré des problèmes que l'on ne connaissait pas en réalisant des projets individuellement. Une bonne organisation du travail est la clé de la réussite dans un tel type de projet où plusieurs personnes doivent travailler ensemble.

CONCLUSION

Finale­ment, nous pen­sons avoir atteint les objec­tifs fixés, puisque notre applica­tion est opé­ration­nelle et permet de créer un modèle dans son ensemble, malgré la présence de quelques lacunes, dont les principales sont l'impos­si­bi­lité de supprimer une méthode et de modifier un objet du modèle.

Cependant, ce projet nous a apporté à chacun une expérience non négligeable concernant le travail en groupe, puisque cela nous a amené à réfléchir à des problèmes que nous ne nous étions jamais posés lorsqu'il s'agissait de conduire un développement individuel. Le travail en groupe constitue certainement une composante essentielle de la vie professionnelle, et ce projet nous permettra sans aucun doute d'aborder le monde du travail de manière plus sereine.

De plus, nous pensons que notre projet est viable et pourrait servir de base à un développement ultérieur afin d'en compléter les possibilités. En effet, la reprise de cette application par d'autres développeurs serait certainement la plus grande récompense à notre travail.

TABLE DES MATIERES

<u>INTRODUCTION</u>	<u>2</u>
<u>I - PRESENTATION DU SUJET</u>	<u>3</u>
<u>I.1 - La modélisation U.M.L</u>	<u>3</u>
<u>I.2 - Pourquoi un générateur de modèles UML ?</u>	<u>5</u>
<u>I.3 - Choix de conception</u>	<u>5</u>
<u>II - PRESENTATION DES FONCTIONNALITES DE L'APPLICATION</u>	<u>6</u>
<u>II.1 Création des différents objets</u>	<u>6</u>
<u>II.1.1 Création d'une classe</u>	<u>6</u>
<u>II.1.2 Création d'un attribut</u>	<u>7</u>
<u>II.1.3 Création d'une méthode</u>	<u>9</u>
<u>II.1.4 Création de liens</u>	<u>10</u>
<u>II.2 Déplacement</u>	<u>12</u>
<u>II.3 Suppression</u>	<u>13</u>
<u>II.4 Protection</u>	<u>14</u>
<u>II.5 Sauvegarde / Restauration</u>	<u>14</u>
<u>III – LIMITES ET EVOLUTION POSSIBLES DE L'APPLICATION</u>	<u>15</u>
<u>III.1 Limites de l'application</u>	<u>15</u>
<u>III.2 Evolutions possibles</u>	<u>16</u>
<u>III.2.1 Les améliorations graphiques</u>	<u>16</u>
<u>III.2.2 Impression du modèle</u>	<u>17</u>
<u>III.2.3 La gestion de plusieurs modèles en même temps</u>	<u>17</u>
<u>IV – ASPECT HUMAIN</u>	<u>18</u>
<u>IV.1 Problèmes de planning</u>	<u>18</u>
<u>IV.2 Problèmes de versions</u>	<u>18</u>
<u>IV.3 Répartition du travail</u>	<u>18</u>
<u>CONCLUSION</u>	<u>21</u>