# PIROser - Parseur UML du Diro

Truong Pham(PHAL29018809)

28 février 2013

**Résumé**

Parseur de diagramme de classes UML fait avec amour et passion.

## 1 Introduction

PIROser est un parseur de diagrammes de classes UML écrit en Java. Ce logiciel possède une interface graphique permettant le chargement d'un fichier UML et de visualiser le résultat du parsing. Le diagramme à traiter doit satisfaire la grammaire BNF présentée dans l'énoncé du TP.

## 2 Execution

Pour executer le programme :

```
ant run
```

Pour executer la suite de test :

```
ant test
```

## 3 Implantation et conception

PIROser est divisé en trois grandes parties. L'interface graphique, le parseur et les différentes composantes d'un diagramme de classes UML.

## 3.1 L'interface graphique

L'interface graphique est entièrement définie dans la classe principale
`Piroser.java`. Chaque composante graphique est initialisée en tant que variable globale privée à la classe, cela facilite la mise à jour de celle-ci lors de l'exécution du programme. Par exemple, il est plus simple et moins coûteux de changer les objets dans une liste que d'en créer une nouvelle avec de nouveaux éléments pour remplacer l'ancienne. On donne une pause bien méritée au `garbage collector`.

Tous les événements tirés lors de l'interaction avec l'interface graphique sont traitées par les différentes sous-classes de `Piroser.java` au lieu d'une fermeture pour chaque objet. Cette décision rend le code plus lisible, selon moi, et permet la réutilisation de fonctions s'il y a des éléments semblables. Comme `Piroser::ComponentsSelectionHandler` qui contrôle la selection des listes `Attributes`, `Methods` et `Subclasses`, ou encore `Piroser::SquarePanel` qui crée quatres `JPanel` avec les mêmes charactéristiques. Les programmeurs n'aiment pas la répétition.

Au lancement du logiciel, le programme appelle la fonction `init()` qui, tout comme son nom l'indique, initialise la création de l'interface graphique et attache des contrôles aux objets créés. Ensuite, PIROser attend que l'usager interagit, ce n'est que lorsque celui-ci appuie sur `Parse` que le programme continue.

Une fois que le parsing est complétée et sans erreur, le logiciel remplie le `JList` de `Class` avec l'information parsée. Après, lors de la sélection d'une classe, on affiche les propriétés de celle-ci dans les listes appropriées. Le champ `details` est utilisé seulement lorsque l'on choisit une aggégation ou une association, car attributs, méthodes et sous-classes sont assez représentatifs.

## 3.2 `Lexer` et `Parser`

Ces deux classes s'occupe du traitement du fichier chargée, ce sont celles-ci qui s'occupent du gros travail. Tout d'abord, le `Lexer` lit chaque caractère du fichier chargée et crée une liste de `tokens`. Chaque `token` est un mot sans espace ou une ponctuation parmi

```
<punctuation> ::= "(" | ")" | "," | ";" | ":"
```

Le lexeur ne détecte pas du tout les erreurs de syntaxe, il ne fait que transformer le fichier à traiter en une liste de mots.

Une fois que `Lexer.java` à terminer son travail, la liste de `tokens` est passé à la classe `Parser.java`. C'est à ce moment que le programme valide le diagramme de classes UML. Chaque partie du parseur traite la grammaire d'un élément de classe UML particulier. En d'autres mots, `parse_class()` s'occupe de valider les `Classe`, `parse_aggregation()` se charge des `Aggregation` et ainsi de suite. Il en est demême pour les sous-composantes comme `Argument` dans un `Operation`. C'est la méthode `parse_declarations()` qui détecte quel élément doit être valider et passe le contrôle à la méthode appropriée. S'il y a une erreur dans la grammaire au moment du parsing, une exception est lancée par `InvalidUMLException`, le programme retourne l'erreur de parsing et il s'arrête. Si le parsing a été un succès, la propriété `model` de `Parser` contient toutes les composantes du diagramme de classes UML qui ont été transformées en objets.

### 3.3 Composante de diagramme UML

Toutes les composantes du diagramme de classe UML sont représentées par des objets. Ces objets sont crées lors du parsing de chaque composante et attachées à l'objet souche `Model`. Les différents objets ont été définis à partir de la grammaire dans l'énoncé du travail. C'est la façon qui me semblait le plus intuitif pour représenter les composantes, et cette décision a semblé me porter fruit car le tout s'harmonise très bien.

## 4 Problèmes et améliorations possibles

La lacune majeure dans cette implantation c'est qu'il n'y a pas de lien explicite entre une `Classe` et ses associations, aggrégations ou généralisations. Pour trouver ces relations, il faut parcourir la liste de chaque élément et la comparer avec le nom de la classe que l'on cherche. Une amélioration possible serait donc d'attacher ces dernières à l'objet `Classe` au lieu de l'objet souche. Par le fait même, on peut détecter l'intégrité du diagramme de classes UML. Pour le moment, on suppose que le fichier chargé est intègre, c'est à dire que toutes les classes nommées dans les associations, aggrégations ou généralisations ont été définies.

La raison pourquoi les classes sont définies par un objet `Classe` au lieu de `Class` dans tout le projet est dû au fait que que le mot

`class` est réservé en `Java`. J'ai donc trouvé un nom sans ambiguïté et le plus semblable possible, en fait j'ai seulement traduit `class` en français. Il y a plus de détails sur cette décision dans les commentaires de `Classe.java`.

# 5 Notes de l'auteur

Je tiens à m'excuser pour les terribles traductions vers le français lors de la lecture de ce texte, spécialement pour tirer un événement au lieu de "fire an event", celle-là m'a vraiment pincé l'estomac. Désolé aussi pour mon code complètement en anglais, j'ai un petit faible pour `read_token()` au lieu de `lire_jeton()`.