QUALITÉ DU LOGICIEL ET MÉTRIQUES - IFT 3913
Conception et Implantation d'un parseur

Travail présenté à
Maude Sabourin
Khady Fall

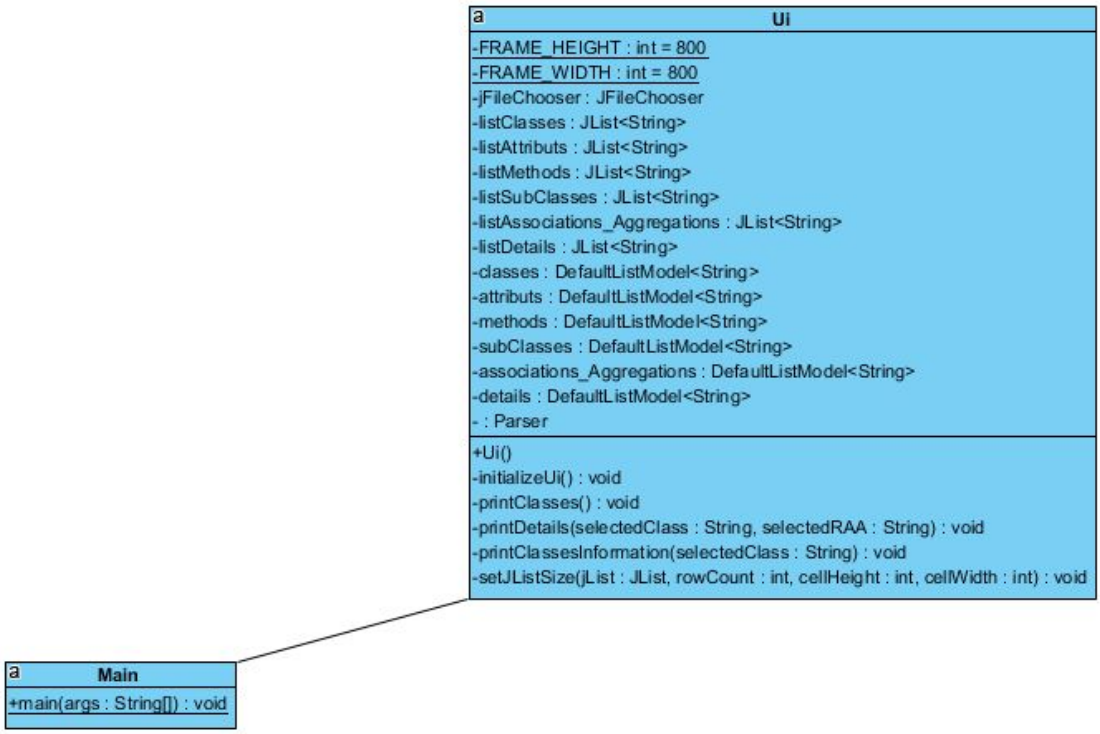
Travail fait par
Sami Steenhaut - 20061630
Eduard Voiculescu - 20078235

Université de Montréal
4 octobre 2018

Dans ce document, l'emploi du masculin pour désigner des personnes n'a d'autres fins que celle d'alléger le texte.

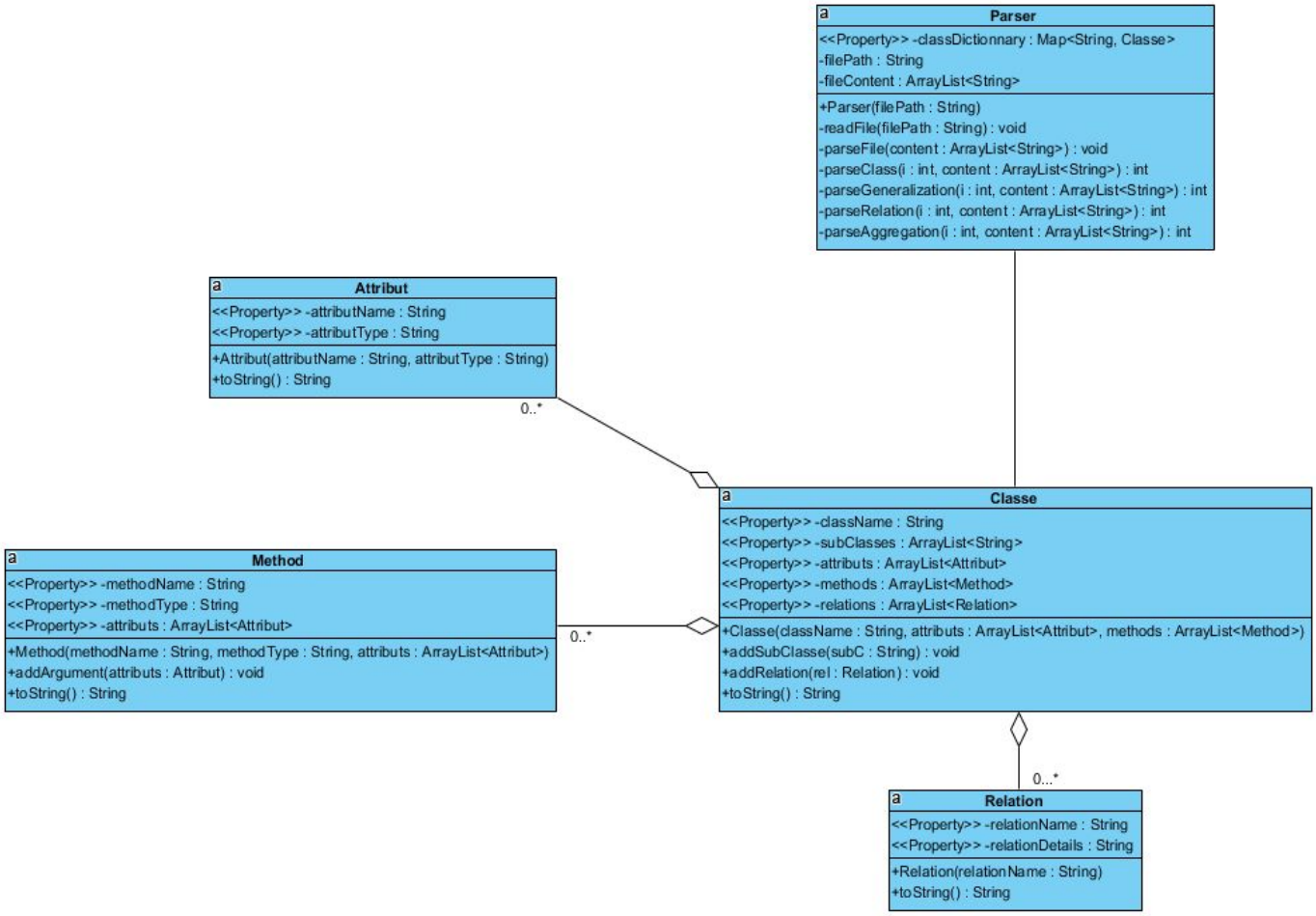
Diagramme de Classe UML

➔ Diagramme de **View**



Dans notre paquet de view, nous avons seulement 2 classes: Main et Ui. Main sert seulement à lancer le programme. Dans notre psvm de notre main, nous l'avons entouré avec un try catch lorsque nous créons l'instance de Ui. Ceci a pour but d'attrapper une erreur s'il y a un problème avec la création de la Ui. Dans notre classe Ui, nous avons décidé de mettre toute l'information nécessaire dans des DefaultListModel que nous attachons à nos JList pour que ces dernières soient visible dans l'interface utilisateur. Il a également un objet Parser qui est créée et qui lie le view au model.

➔ Diagramme de **Model**



Pour notre paquet **model**, on constate qu'il y a plusieurs classes. Comme nous allons le mentionner plus bas, nous avons pris une approche modulaire pour ce projet. S'il y avait possibilité de créer des objets nous le faisions (sans exagération bien évidemment). Nous avons profité grandement des ArrayList<__*__> puisque ces dernières sont très facile à manipuler, facile à extraire l'information et facile à ajouter l'information. Nous avons également implémenté des toString() spécifique à chaque classe respectivement ce qui sera visible pour l'utilisateur dans l'interface graphique.

Description des modules

Pour ce projet, nous avons opté pour une approche MVC (Model View Controller). D'une part c'est une manière optimale de programmer et d'une autre part c'est très facile pour la distribution des tâches. De plus, les interactions entre les modules ("Model" et "View") sont limités. Cela nous permet de faire un changement quelconque dans le "Model" sans impacter (ou un impact très minimaliste) le module "View" et vice versa. Aussi, pour le debugging, puisque les modules sont séparés, c'est beaucoup plus facile pour trouver nos bug.

Dans le paquet **View**, il y deux classes. Plus spécifiquement Main.java et Ui.java. La classe Main.java sert pour partir le programme. La classe Ui.java va tout créer l'interface utilisateur. Nous avons suivi presque à la lettre le design qui était indiqué dans la section 6 de la donnée du TP. De plus, pour le "C" de MVC (le contrôleur), nous n'avons pas opté de créer un paquet à lui-même puisque le contrôleur est géré par le bouton pour trouver le fichier adéquat et par les clics de JList (qui montre plus d'informations si nécessaire) sur les classes et sur les associations and aggregations.

Le paquet **Model** est le paquet contenant cinq classes: Parser.java, Attribut.java, Classe.java, Method.java et Relation.java. Le parseur est assez simple. Premièrement, nous avons décidé de mettre les classes (contenant les informations nécessaires) dans un HashMap. La raison principale pour laquelle nous avons choisi un HashMap au lieu d'un ArrayList (puisque nous en avons utilisé beaucoup partout à travers notre programme) est que lorsque nous voulons parcourir cette dernière, il est beaucoup plus rapide et efficace d'aller chercher la clef directement au lieu de parcourir un ArrayList à chaque fois. Le parseur passe à travers le fichier à lire (fichier .ucd) et met le tout dans un ArrayList file Content. Lorsqu'il met les lignes dans le file Content, il y a des vérifications générales de grammaire d'UML qui sont faites. Ces vérifications de grammaire d'UML sont pour s'assurer que le fichier commence avec MODEL <nom du modèle>, que les ::= sont présentes et qu'elles peuvent être de différentes combinaisons possibles (:, ::, =, :=, ::=) et bien d'autres. Puisque nous savons pas exactement ce que la déclaration d'un attribut, d'une opération et bien d'autres, nous avons laissé la liberté que la déclaration soit n'importe quelle de ces combinaisons ci-haut.

De plus, notre programme est très modulaire. Nous avons créée beaucoup de classes pour vraiment bien séparer les différentes parties du programme. Nous avons choisi de faire un décomposition approfondie de nos classes. Non seulement c'est un bonne pratique de programmation mais également c'est très facilement maintenable pour les développeurs.

Maintenant, face à notre interface utilisateur (Ui.java), nous avons opté d'utiliser la librairie "Swing", car c'est une libraire facile à utiliser. Il a également le fait que "Swing" est très riches en fonctionnalités et la documentation est claire et facile à comprendre.

RÉUTILISABILITÉ

Un autre point que nous aimerions soulever est la réutilisabilité. Puisque notre programme est séparé en classes et très modulaire, il est assez simple d'extraire une classe respective et la rendre réutilisable à fin d'utilisation pour d'autres projets, voire même d'autres langages. Nous nous sommes vraiment forcé à faire le plus possible en orienté objet question de réutilisabilité et de facilité de débogage. Et bien évidemment, c'est également une bonne pratique.

ROBUSTESSE

Par la suite, nous allons vous parler de la solidité de notre programme. Comme mentionnée plus haut, nous avons fait quelques vérifications sur la grammaire BNG de diagrammes de classes UML (section 5 de la donnée du TP). À plusieurs endroit dans notre parseur nous avons essayés du mieux qu'on peut pour que notre parseur convienne à la syntaxe BNF. Il se peut qu'il y ailles certains "use case" qui ne fonctionnent pas. Nous avons fait des tests, mais bien évidemment, comme nous ne savons pas comment vous allez tester notre programme nous avons imaginé plusieurs scénarios et plusieurs exemples de cas non acceptable et avons essayé de faire en sorte que le parseur répond à cela.

De plus, nous avons également respecté la contrainte de la lecture du fichier approprié qui a été soulevé de nombreuses fois dans les discussions sur la platform universitaire StudiuM. Il va de soit que notre programme ne voit même pas les fichier qui sont d'un autre type que ceux ayant comme suffixes .ucd.

D'autre part, nous avons également ajouté une multitude de "try" et "catch" partout à travers notre code. Nous les avons surtout ajouté aux places qu'il est le plus susceptible d'avoir un erreur générée (Null Pointer est un des cas probable: un HashMap ou un ArrayList vide).

MAINTENABILITÉ

Comme la définition de maintenabilité donnée en classe: “Ensemble des attributs portant sur l’effort nécessaire pour faire des modifications données”, notre programme est facilement maintenable. D’une part il est facilement analysable, puisque comme nous avons mentionné, nous avons séparé le tout en MVC et nous avons également créer plusieurs classes. Si jamais il y a un problème quelconque, régler ce dernier est assez facile dû à cette modularisation de classe.

Il va de soit que c’est pareil pour la facilité de modification. Le seul gros problème qui peut arriver, puisque dans notre parseur, nous envoyons un HashMap<String, Classe> à notre view, est s’il y a un gros changement à la structuration du dictionnaire, alors il se peut qu’il y aille des problèmes entre le model et le view. Par contre, s’il y a des changement au sein de Classe, il ne devrait pas y avoir un impact, tant qu’on ne change pas de manière drastique le constructeur respectif et le nom des variables (qui ne devront pas avoir eu lieu sans justification claire).

TESTS

Dans le paquet **src**, nous avons également un paquet nommé **test**. Dans ce paquet, il a une classe Test.java qui contient une panoplie de tests. Ces tests ont pour but de tester une variété de bogue possible que nous avons géré dans notre parseur.

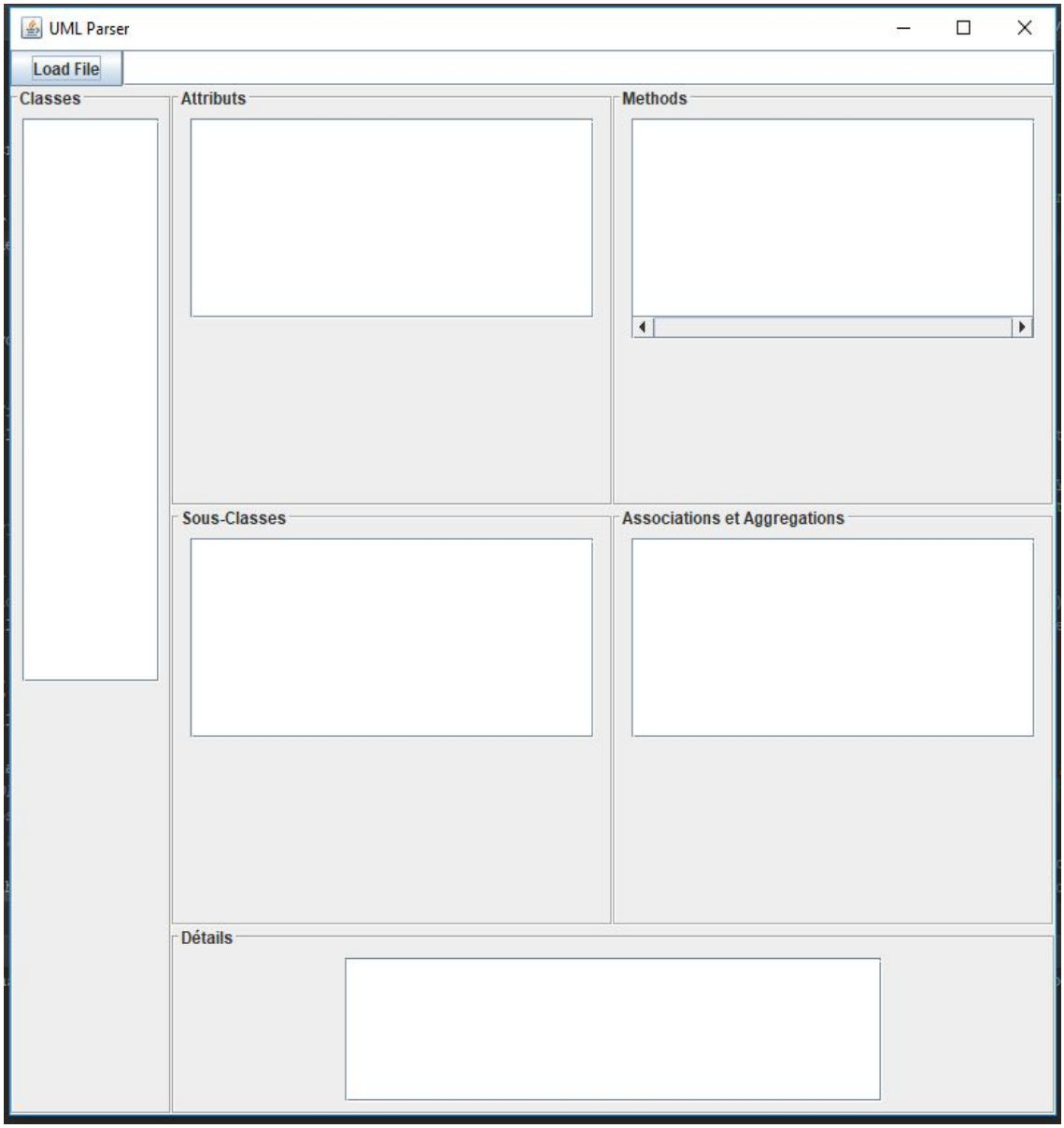
De plus, il y a un de nombreux fichiers nommant *ligue.ucd et ligue*.ucd avec de différents diagrammes de classes UML. Comme mentionné plus haut, puisque nous ne savons pas sur ce que nous allons être testé exactement, nous avons essayé de corriger des bogues en relation avec la grammaire BNF et une variété d’autres bogues que nous pensions qui pourrait arriver.

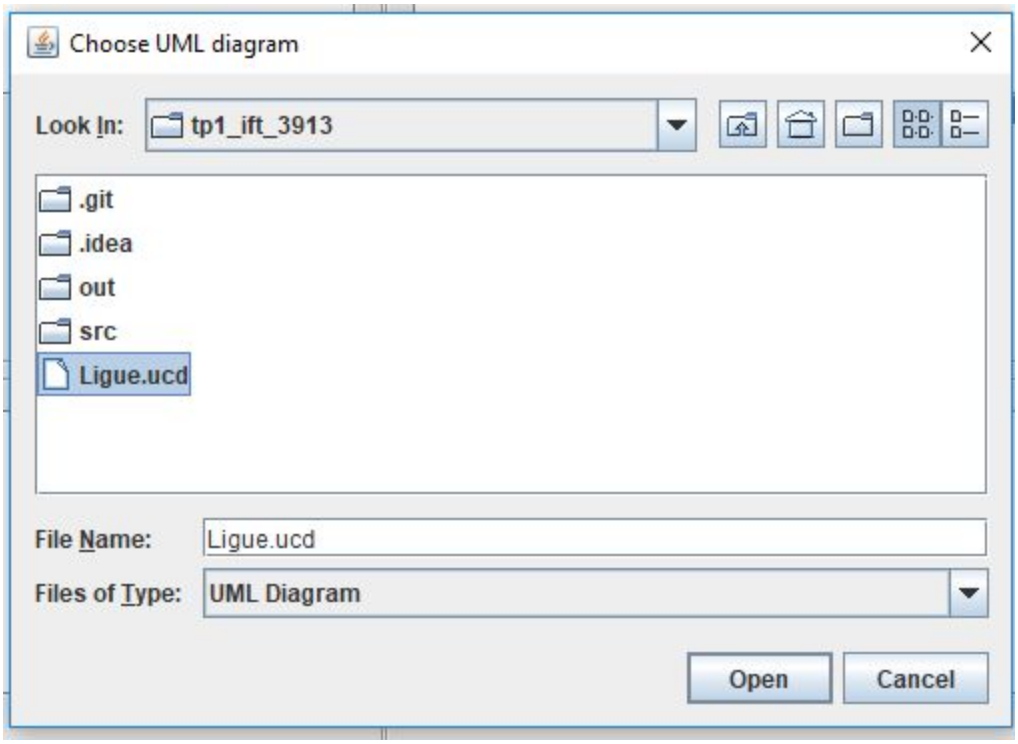
MANUEL D'UTILISATION

Ce manuel d'utilisation n'a d'autres fins que pour montrer les étapes du manuel d'utilisateur.

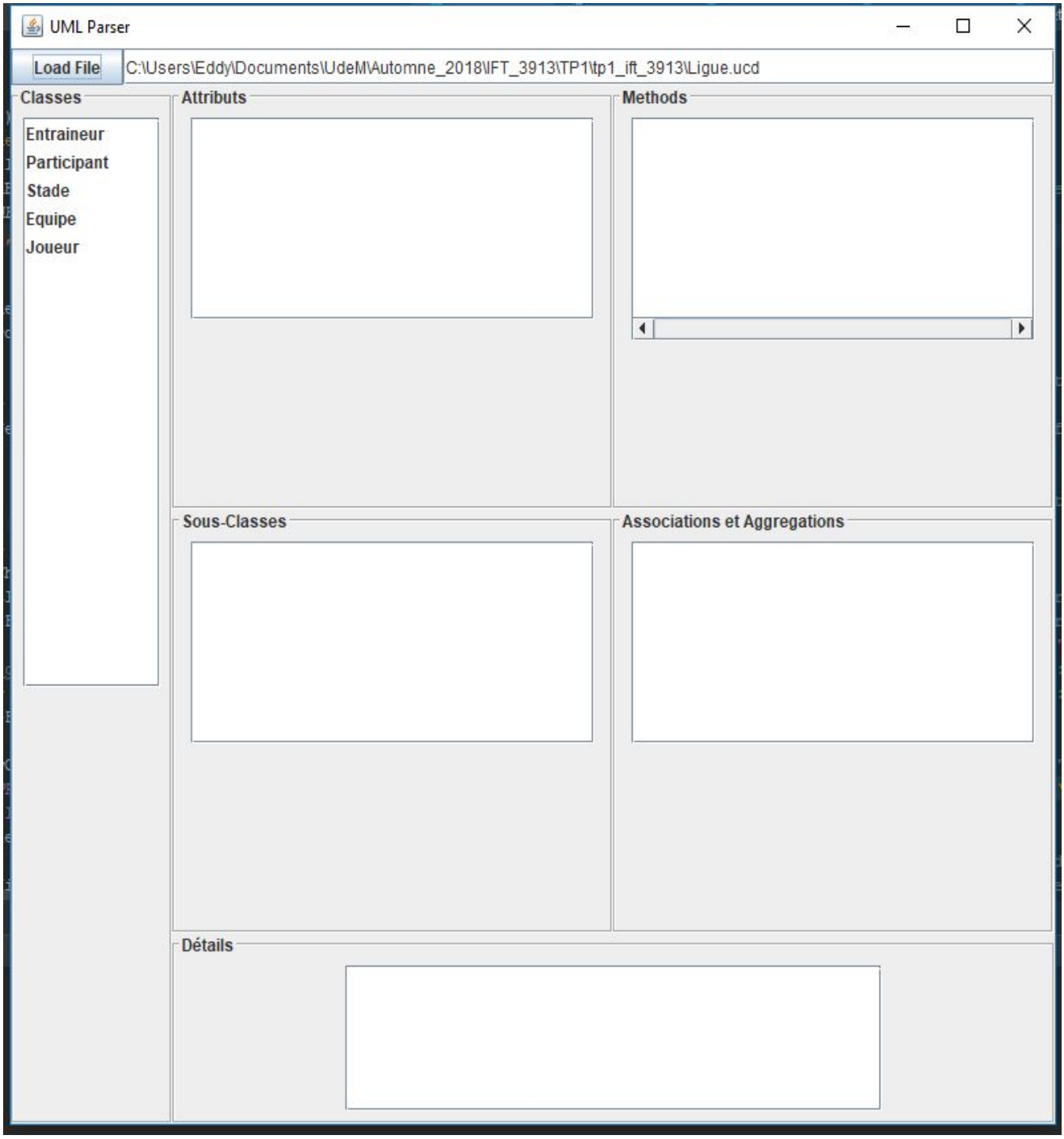
1. Lancer le programme via un double clic sur le fichier .jar ou en écrivant comme commande dans un terminal ou un command prompt:
    a. javac view\Main.java
    b. java view.Main

2. Une fois l'interface lancée: appuyez sur "Load File" et chargez un fichier .ucd et appuyez sur "Open".

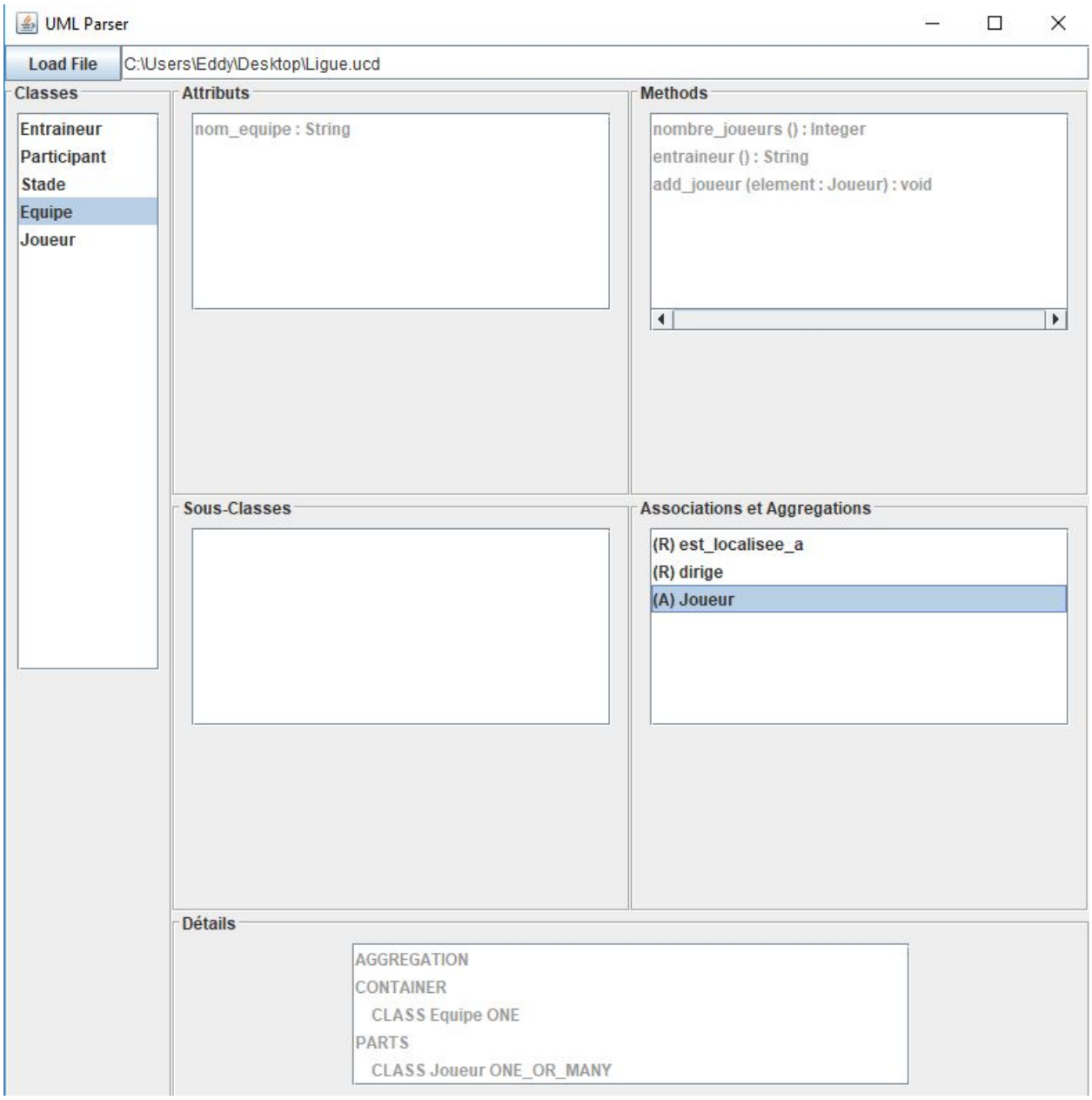



Choose UML diagram
Look In:
tp1_ift_3913
.git
.idea
out
src
Ligue.ucd
File Name:
Ligue.ucd
Files of Type:
UML Diagram
Open
Cancel

3. Une fois le fichier chargé dans l'interface utilisateur, vous serez en mesure de voir le chemin où vous avez choisi votre diagramme de classe. Nous avons décidé de laisser tout le chemin pour que l'utilisateur sache exactement d'où le fichier .ucd provient. Voilà l'interface qui vous fera la bienvenue.



4. Par la suite, l'utilisateur peut s'amuser à cliquer sur une classe et voir de l'information apparaître devant lui. Il y a certain champ qui sont cliquable et d'autres qui ne le sont pas. Cela est allé à la discrétion et à la créativité des développeurs. Voilà l'interface que l'utilisateur verra.



Pour finir, nous avons essayé de rendre l'interface la plus conviviale possible. Il se peut qu'elle ne soit pas le plus belle point de vue design, mais l'utilisabilité en fait parti à plein coeur.