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

Travail présenté à
Maude Sabourin
Khady Fall

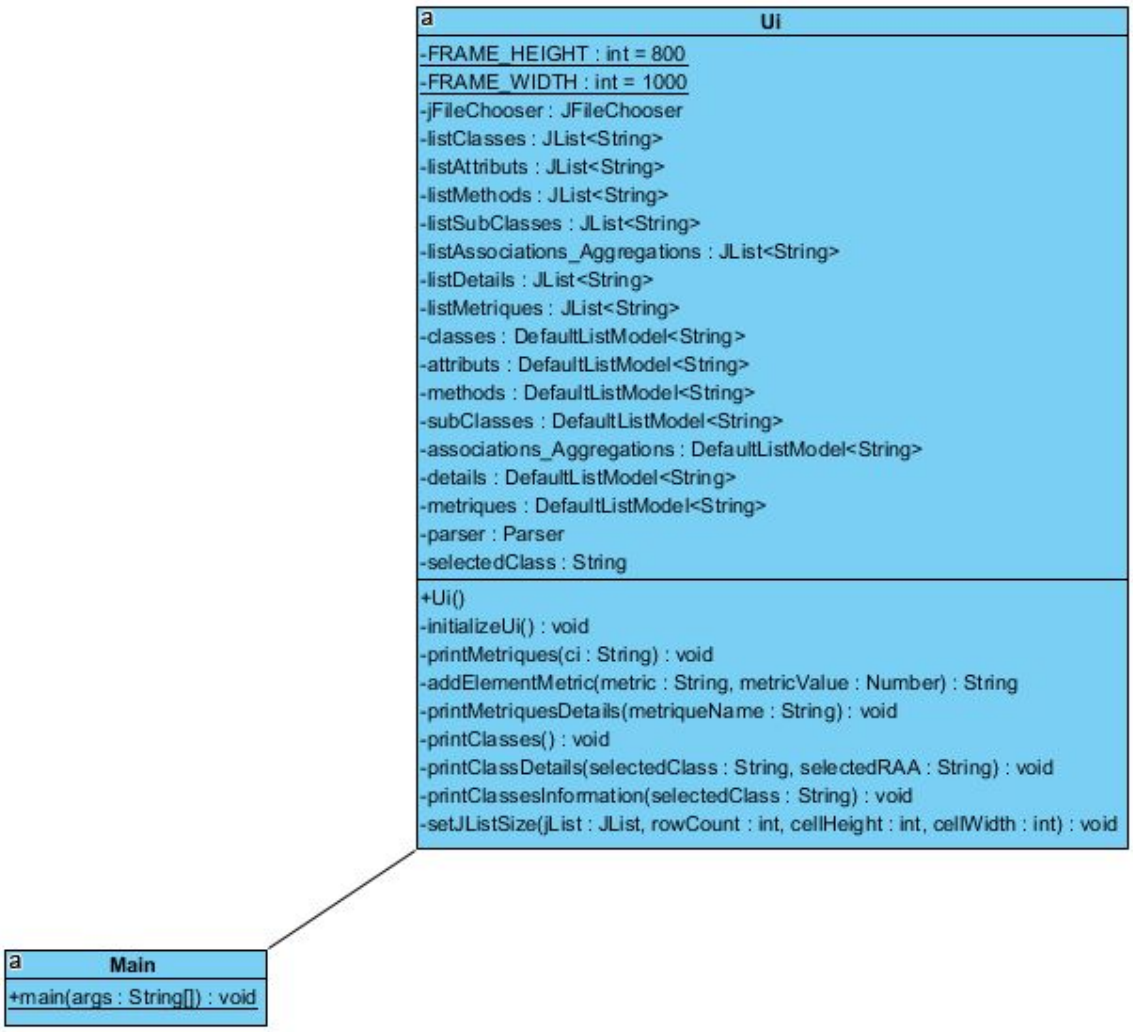
Travail fait par
Sami Steenhaut - 20061630
Eduard Voiculescu - 20078235

Université de Montréal
7 novembre 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

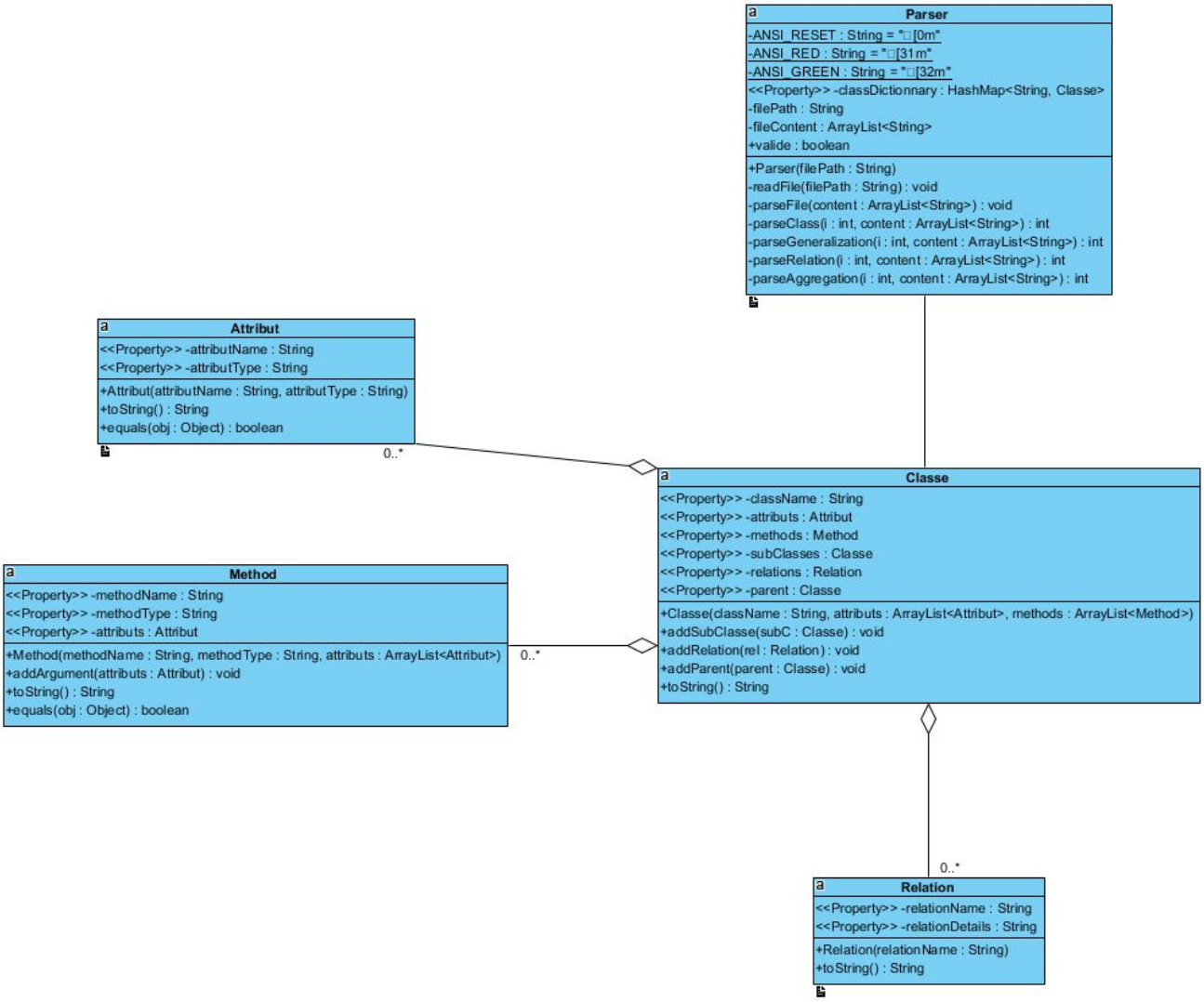
➔ Diagramme de **View**



(Du copié-collé venant du tp1)

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. Ce que nous avons ajouté de plus est la String **selectedClass** pour garder trace sur la classe sélectionnée.

➔ Diagramme de **Model**

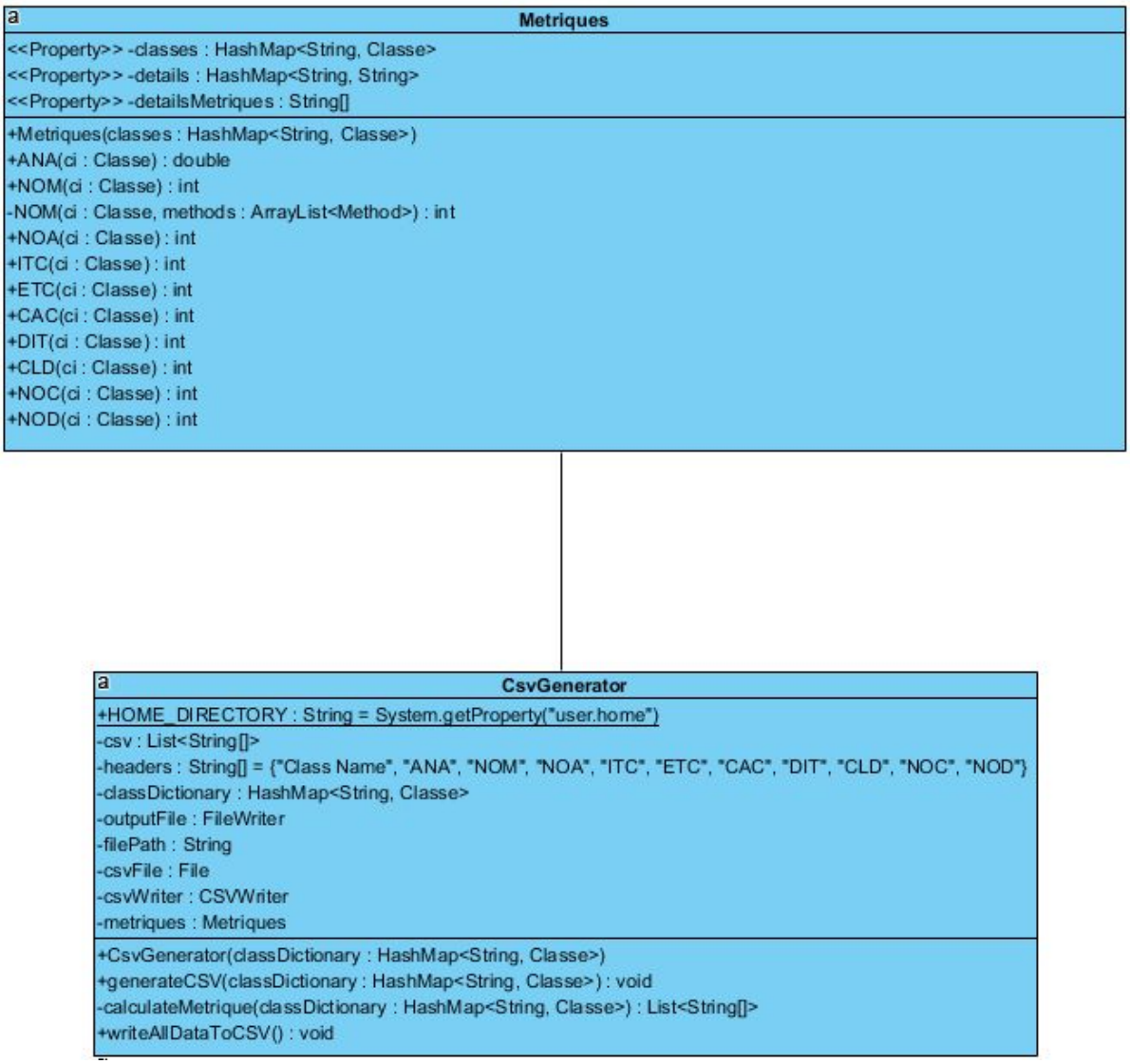


(Du copié-collé venant du tp1)

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. Ce que nous avons ajouté est le boolean valide qui est par défaut true et devient false si une erreur de syntaxe est détectée par le parseur. L'erreur sera affichée sur le command prompt.

Prenez note: Les valeurs de ANSI_RESET, ANSI_RED ET ANSI_GREEN sont respectivement "\u001B[0m", "\u001B[31m", "\u001B[32m" mais Visual Paradigm considérait nos entrées comme une erreur de syntaxe.

➔ Diagramme de **Metriques**



Pour notre paquet **metriques**, nous avons créé deux classes: Metriques.java et CsvGenerator.java. Nous sommes resté dans l'orienté-objet pour ce paquet également. Malheureusement, ce paquet est moins réutilisable que le paquet Model puisque ce paquet est requis de faire des fonctionnalités propres au projet. Donc, il y a perte de *réutilisabilité* dans ce paquet. C'est surtout le cas pour CsvGenerator.java, puisque ce dernier doit faire sortir un fichier csv très spécifique. Le CSV contenant toutes les métriques sera mis sur votre desktop d'ordinateur. Une message sur le command prompt indiquera que le fichier a été généré avec succès.

DESCRIPTIONS DES MODULES

Pour ce projet, nous avons continué sur le cheminement modulaire que nous avons établi lors du premier Travail Pratique (tp). Lors du dernier tp, nous avons programmé en MVC (Model View Controller). Puisque nous avons fait une très bonne modularisation de nos classes et les différentes fonctionnalités, ajouté le paquet **métriques** nous a pas causé de grand problème. Nous avons gardé la même idée. Notre paquet **métriques** contient seulement deux classes se nommant Metriques.java et CsvGenerator. La classe Metriques.java contient toutes les fonctions nous permettant de calculer les 10 métriques nécessaires. De plus, nous avons programmé tous nos tests en TDD (Test Driven Developement), nous allons plus en parler dans la section TEST. Pour la classe CsvGenerator.java, nous avons décidé de la garder dans le même paquet que Métriques.java puisque cette dernière est *dépendante* des métriques. La description des paquets **View** et **Model** est la même que sur le rapport du premier tp. De plus, prenez note que les fonctions pour calculer les métriques de notre classe Métriques.java sont tous en majuscules. Nous sommes au courant que la convention des majuscules en java est réservée pour les variables finales, mais nous les avons laissé en majuscule question de lisibilité. Le tout est pour rendre cela plus lisible et rester “Zen”.

RÉUTILISABILITÉ

La réutilisabilité reste la même que celle présenté au rapport du premier TP. Malheureusement, comme mentionné plus haut, le nouveau paquet **metriques** n’est pas réutilisable. La raison est que les classes ont des fonctionnalités très particulières et précises à ce travail pratique.

ROBUSTESSE

Faisant suite à la solidité du programme du tp1, notre programme est encore très solide. Puisque nous avons rajouté une vingtaine de test, nous avons vraiment testé les cas limites de notre programme. Que ça soit des classes vides, un fichier ne contenant pas de MODEL, un fichier avec toutes les classes doublés et bien d’autres scénarios, nous avons imprimé dans le command prompt une variété de message avertissant l’utilisateur de quelconque problème rencontré. De plus, nous avons fait des tests avec les erreurs probables qui nous viennent en tête et concernant les fonctionnalités ayant le plus de probabilité de briser l’interface utilisateur. Il y a plusieurs try catch statements à certains endroits étant le plus propices à causer des erreurs. Si une erreur est introduite, le programme ne quittera pas, il y aura une impression du stack trace dans le command prompt et un message d’erreur.

Pour de futures projets de programmation, il serait important de soit importer un logger ou même de s'en créer un personnalisé. Ce logger pourrait créer trois fichiers activity.log, debug.log et error.log, chacun ayant un rôle spécifique. Le activity.log présentera l'information des différentes actions faites par l'utilisateur (nom du fichier choisi, boutons cliqués, classes cliquées et bien d'autres). Le debug.log servira seulement pour les programmeurs afin d'imprimer tout ce que le programme fait (activity.log + les entrées de fonctions et bien d'autres). Finalement, le error.log aura pour but de montrer les erreurs rencontrées. Avec le debug.log et le error.log, cela permettra plus facilement aux programmeurs de faire des patch et de porter des changements à certaines fonctions du programme. Le tout pour rendre l'expérience de l'utilisateur facile et agréable.

MODIFICATIONS APPORTÉES

Il y a quelques modifications que nous avons dû apporter. En premier lieu, nous avons ajouter une variable se nommant **selectedClass** pour nous aider à garder trace de quelles métriques que nous voulons calculer. Puisque lorsque l'utilisateur appuie sur une classe, nous enlevons toutes les anciennes informations des métriques (si ces dernières ont été calculées) nous devons garder trace de la classe sélectionnée. De plus, notre programme gère l'héritage multiple. Nous savons que l'héritage multiple n'est pas toléré dans plusieurs langages de programmations, nous avons toutefois laissé les portes ouvertes pour ce dernier puisqu'il y a quand même quelques langages qui le tolère.

En deuxième lieu, suite à la lecture de plusieurs questions sur le forum de l'UdeM et de Slack, nous avons vue qu'il était mentionné de laisser le retour du constructeur d'une classe vide. Comme il n'est pas mentionné nul part dans la grammaire BNF dans la donnée du tp1, nous avons décidé que le retour d'un constructeur s'intitulerait *Constructor*. La raison principale est pour vraiment différencier avec les autres méthodes. Comme ça lorsque nous lisons les méthodes, nous verrions le type de retour *Constructor* et nous saurons automatiquement que c'est le constructeur de la classe. Avant même que vous dites : "Mais qu'est-ce qui se passe si quelqu'un crée une classe *Constructor* ?", en effet cela serait un bogue nécessitant un patching. N'oublions pas que le constructeur en Java est supposé toujours être la première "méthode" après la déclaration de toute les propriétés et/ou variables d'une classe quelconque. Cela pourrait aider à la lisibilité.

DIFFICULTÉS RENCONTRÉES

Par la suite, nous avons rencontrées plusieurs difficultés lors de l'implémentation du tp. Premièrement, lors du premier tp, nous avons utilisé une ArrayList<String> comme structure de donnée pour les sous-classes d'une classe. Cela nous satisfaisait pour afficher une sous-classe sur l'interface utilisateur. Malheureusement, pour le calcule de plusieurs métriques, plus précisément les métriques ayant besoins d'accéder les sous-classe, nous avons besoin de l'objet de classe au complet. Les métriques en questions sont CLD et NOC. Alors nous avons fait un petit changement dans notre classe Classe.java en remplacant notre structure de donnée de ArrayList<String> par ArrayList<Classe>. Cela nous a permi de bien calculer les métriques qui nécessitait d'aller visiter les sous-classes d'une classe.

Ensuite, nous avons bloqués pendant un certain temps à bien implémenter le calcul de la métrique NOM. La raison est que nous regardons si une liste de méthods contient une certaine méthode. Le problème est que nous passons par ArrayList<Method>.contains(method) et le contains passe par le obj1.equals(obj2) de Object.java. Malheureusement comme vous pouvez voir, puisque utilisons notre propre object (Method), nous devons redéfinir equals() pour la classe Method. De plus, en comparant les méthodes, nous devons comparer les attributs également, nous étions également obligé de redéfinir .equals() pour notre classe Attribut. Une fois que nous avons redéfini .equals() de manière appropriée aux deux classes respectives, nous étions en mesure de bien calculer la métrique NOM pour une classe. Au début, avant même de redéfinir .equals() deux fois, nous avions écrit le tout dans la fonction de NOM dans la classe Metriques.java. Mais en relisant la donnée du tp, il est indiqué de "surcharger les toString()" alors nous avons décidé de faire la même chose pour les .equals(). Bien évidemment, le tout est pour que le code reste beau et élégant. Simplicité et lisibilité avant tout!

Une autre difficulté rencontré a été de créer les AGGREGATION et les RELATION. Après de nombreuses lecture en ligne contradictoires et certaines affirmant ce que nous avons vue au cours IFT 2255, nous avons essayé d'entamer cette dernière tâche. Pour commencer, nous avons créé les AGGREGATION et les RELATION en regardant notre diagramme de classes UML. Puisque nous avons eu de la misère à distinguer bien la différence entre la relation et l'agréation (nous nous sommes contenté sur la définition de : https://softwareengineering.stackexchange.com/questions/176049/what-is-the-use-of-association-aggregation-and-composition mentionnant qu'une agrégation est qu'une classe qui ne peut vivre sans les autres et une relation est une classe qui utilise une instance d'un autre objet). C'est peut-être pas la définition parfaite, mais nous nous sommes contentés de cette dernière.

TEST

En continuant sur les mêmes lignes que le tp1, nous avons ajouté plus de tests à notre classe Test.java. En premier lieu, nous avons fait une fonction retournant un booléen s'intitulant **metricTesting** qui a pour but de comparer la valeur des métriques d'une classe avec les métriques calculées (entrée manuelle dans les arguments vs. le retour des fonctions des calcules des métriques). Par la suite, nous avons utilisés cette fonction pour tester de nombreux fichiers Ligue*.ucd. Pour nommer quelques exemples : "LigueMultipleParentsMetric", "LiGueMultipleParentSameLevelMetric", "LigueMetric" où toutes les classes sont doublées et bien d'autres. Nous avons également fait un test par métrique (les fichiers *.ucd en conséquence). Prenez note que tous les résultats sont imprimés dans la ligne de commande avec les erreurs que le parseur à vue (les erreurs imprimées du parseur n'arrêtent pas le fonctionnement du programmes).

Pour l'implémentation des métriques, nous sommes procédés en TDD (*Test Driven Development*). La raison principale pour laquelle nous avons décidé de faire cela est pour nous sauver du temps. Au lieu de toujours lancer la GUI, choisir un fichier quelconque et comparer les résultats, ou même d'avoir un mur rempli de Post It avec les valeurs de métriques, nous nous sommes compté sur l'invention du fameux ingénieur logiciel *Kent Beck - Test Driven Development*. Nous avons commencé par créer 10 fichiers s'intitulant **Model_métriqueàtester_Test_metric.ucd**. Par la suite, nous avons créé 10 tests (1 pour chaque métrique) avec les résultats attendus. Comme en TDD, nous avons écrit des tests qui ne passent pas, et nous devons les faire passer au fur et à mesure. Étonnamment, procéder cette manière nous as sauvé beaucoup de temps. Une fois tous nos tests verts, nous sommes étions assez confiant de ne plus toucher aux fonctions pour calculer les métriques puisque ces dernières fonctionnaient bien avec les Use Case que nous avons inventés. Bien évidemment nos tests couvrent les cas possibles que nous avions pensé, il se peut très bien qu'il existe d'autres cas extrêmes résultant à des bogues. Le tout nécessitant un patching dans le futur!

De plus, lorsque nous étions en train de convertir notre code en fichier .ucd, nous avons réalisé qu'en retournant un HashMap<String, Classe> il y avait des problèmes. C'est un bug qui nécessite du patching. Pour palier à ce bogue, si une méthode retourne un HashMap<String, Classe> ou un attribut est de type HashMap<String, Classe> (c'est le retour d'un HashMap en tant que tel qui causait problème) nous faisions un retour de HashMap<String.Classe>. Cela reste claire pour les utilisateurs et il n'y as pas de perte d'information.

ANALYSE DU CODE

	Chemin	Nom	Taille (# classes)	Taille (NCLOC)	Taille (CLOC)	ANA	NOM	NOA	ITC	ETC	CAC	DIT	CLD	NOC	NOD
	src\model	Attribut	5	34	38	0.714286	7	2	0	1	3	0	0	0	0
	src\metriques	Metriques	2	174	125	0.888889	18	3	11	0	2	0	0	0	0
	src\model	Relation	5	25	32	0.5	6	2	0	1	1	0	0	0	0
	src\view	Ui	2	308	82	1.222222	9	19	0	0	0	0	0	0	0
	src\Test	Test	1	361	154	2.409091	22	25	1	0	0	0	0	0	0
	src\model	Classe	5	67	75	0.705882	17	6	3	12	5	0	0	0	0
	src\model	Method	5	68	53	0.8	10	3	1	0	3	0	0	0	0
	src\metriques	CsvGenerator	2	57	38	0.75	4	9	0	0	1	0	0	0	0
	src\view	Main	2	12	3	0	0	0	0	0	0	0	0	0	0
	src\model	Parser	5	221	78	1.375	8	7	0	0	1	0	0	0	0
Minimum			1	12	3	0	0	0	0	0	0	0	0	0	0
Maximum			5	361	154	2.409091	22	25	11	12	5	0	0	0	0
Moyenne			3.4	132.7	67.8	0.936537	10.1	7.6	1.6	1.4	1.6	0	0	0	0
Mode			5	#N/A	38	#N/A	#N/A	2	0	0	1	0	0	0	0
Médiane			3.5	67.5	64	0.775	8.5	4.5	0	0	1	0	0	0	0
Moyenne Géométrique			2.950509385	80.4277313	48.21159225	0	0	0	0	0	0	0	0	0	0
Moyenne Harmonique			2.5	46.2515302	20.45302169	0	0	0	0	0	0	0	0	0	0
Écart Type			1.712697677	125.828852	45.3818368	0.638423	6.854844	8.167687	3.438346	3.747592	1.646545	0	0	0	0

Suite à la génération du fichier CSV et l'ajout de quelques informations voilà le résultat final de notre code. Comme on peut voir, il y a beaucoup de zéros dans les dernières colonnes (DIT, CLD, NOC et NOD), la raison est que nous n'avons aucun héritage dans notre projet. Nous aimerions également que vous portiez attention à la moyenne de NCLOC et CLOC. On constate qu'il y a environ 35-40% des lignes qui sont des lignes de commentaires. Nous avons essayé de suivre les bonnes pratiques de documentation de la Javadoc et nous avons commentés lorsqu'il était nécessaire. À certaines places, le mode est un **#N/A**, c'est puisqu'il n'a pas de mode.

Il y a plusieurs mesures de calculs qui ne font pas de sens prenons comme exemple la Tailles (# classes). Vue que c'est une mesure de type nominale, la moyenne, la médiane, la moyenne géométrique, la moyenne harmonique et l'écart-type n'apportent aucunes informations pertinentes.

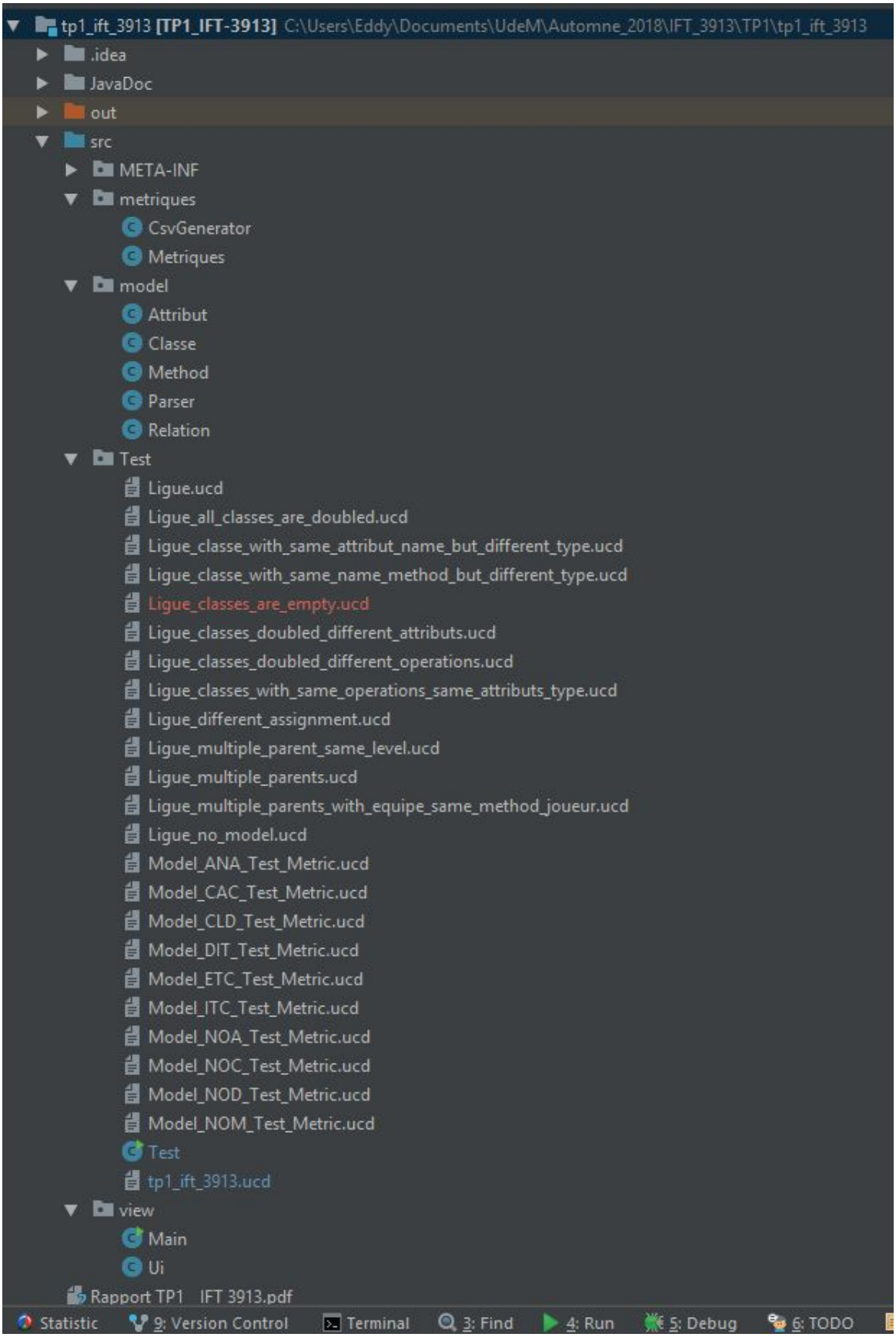
De plus, il y a plusieurs mesure de tendance centrale qui ne sont pas pertinentes pour nos métriques. Sauf pour le minimum, le maximum et la moyenne, les autres mesures de tendance centrales n'apportent pas vraiment d'information pertinentes.

Nous avons essayé de faire l'analyse de code la plus concise possible sachant que nous avons une limite de 10 lignes environs (même si nous en avons 15 …).

TEMPS APPROXIMATIF

Nous avons dédié environ 7 heures chaque.

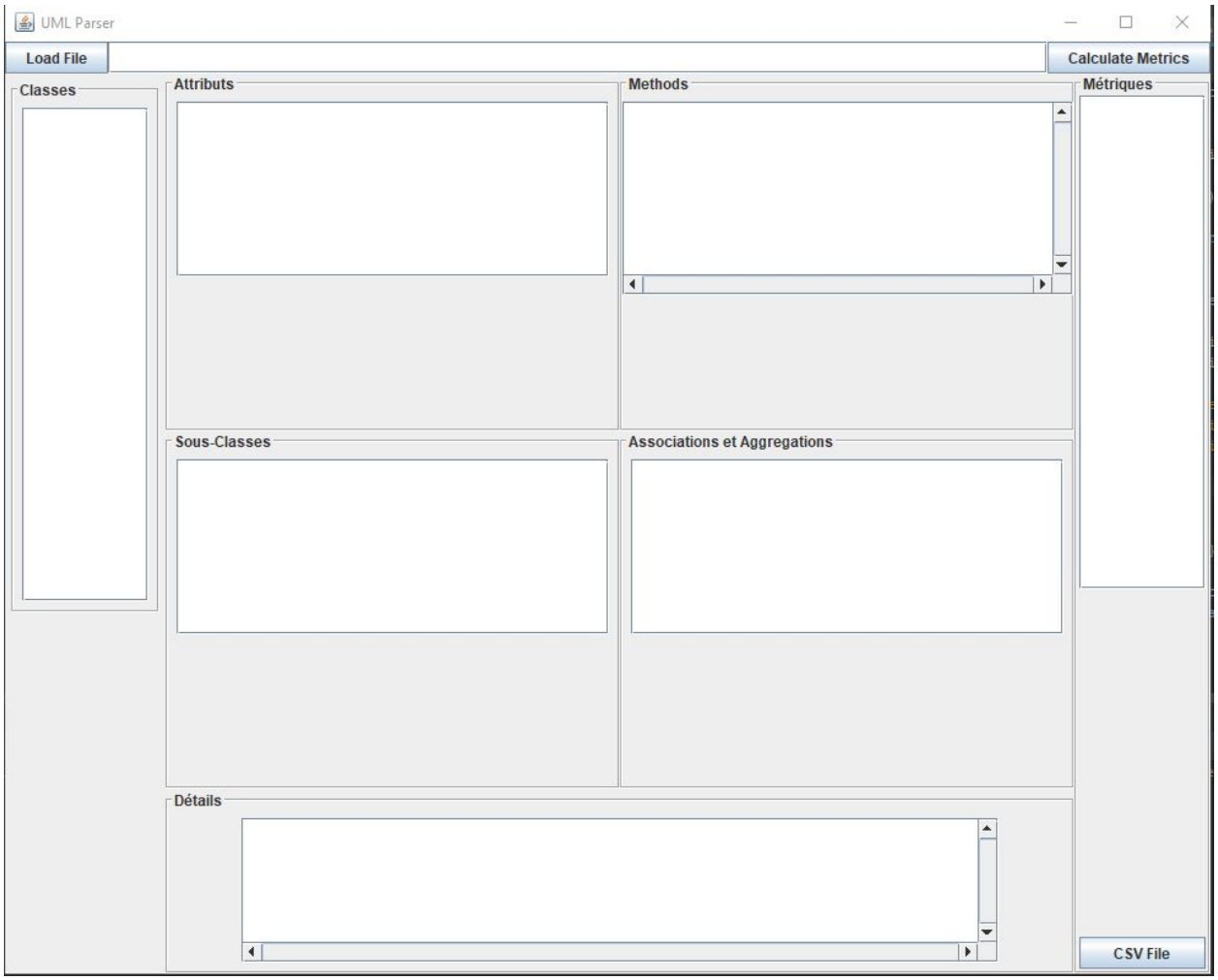
Voilà l'arborescence de notre code au complet. On peut bien voir les 4 paquets (metriques, model, Test et view).

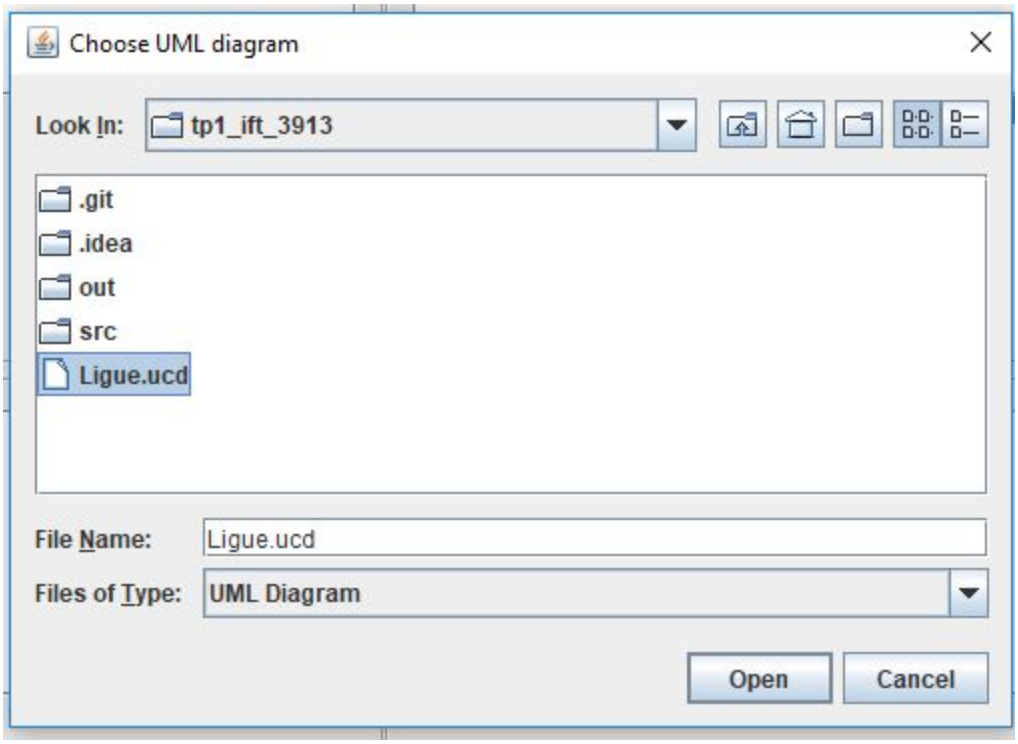


MANUEL D'UTILISATION

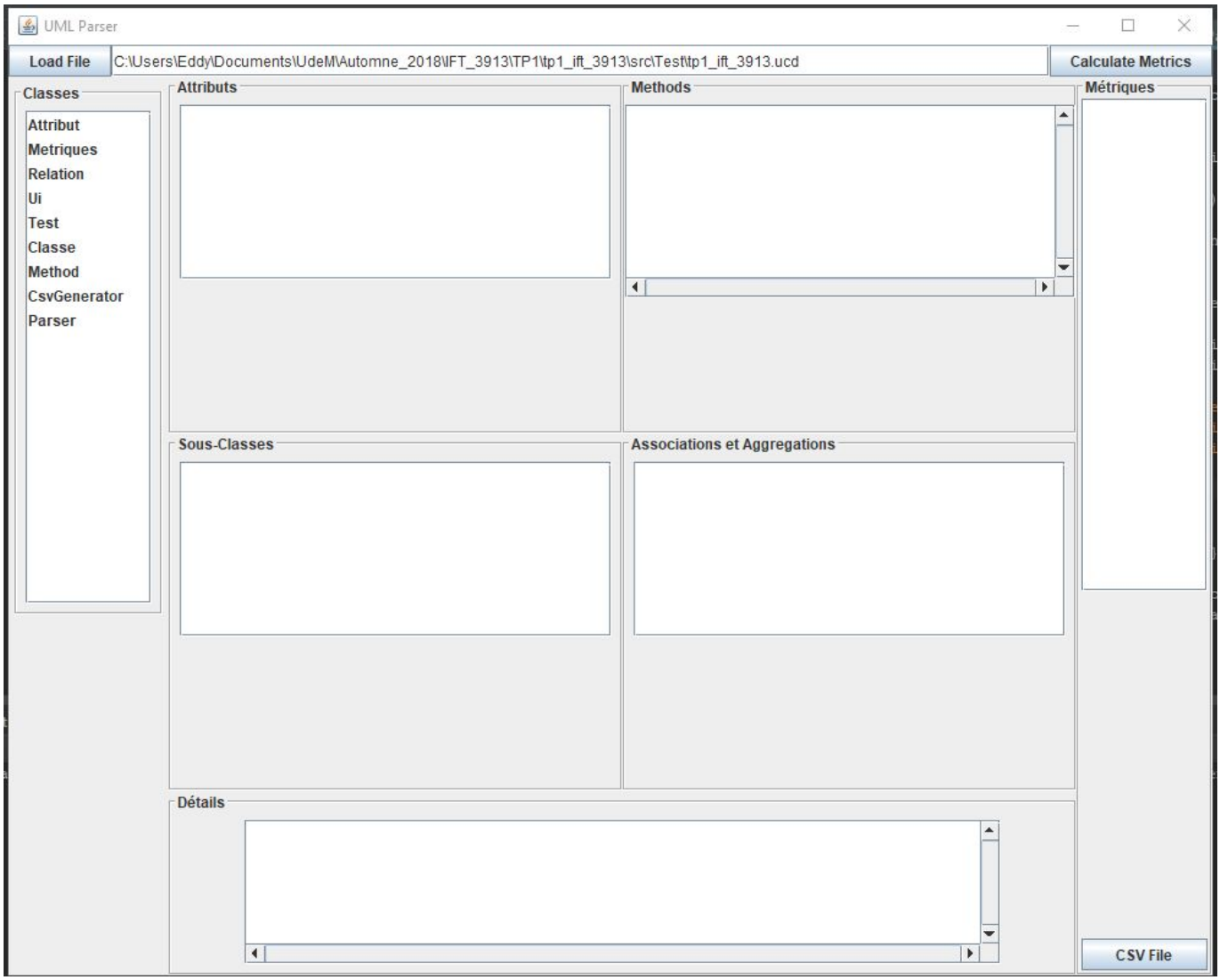
Ce manuel d'utilisation n'a d'autres fins que pour montrer les étapes du manuel d'utilisateur.

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".

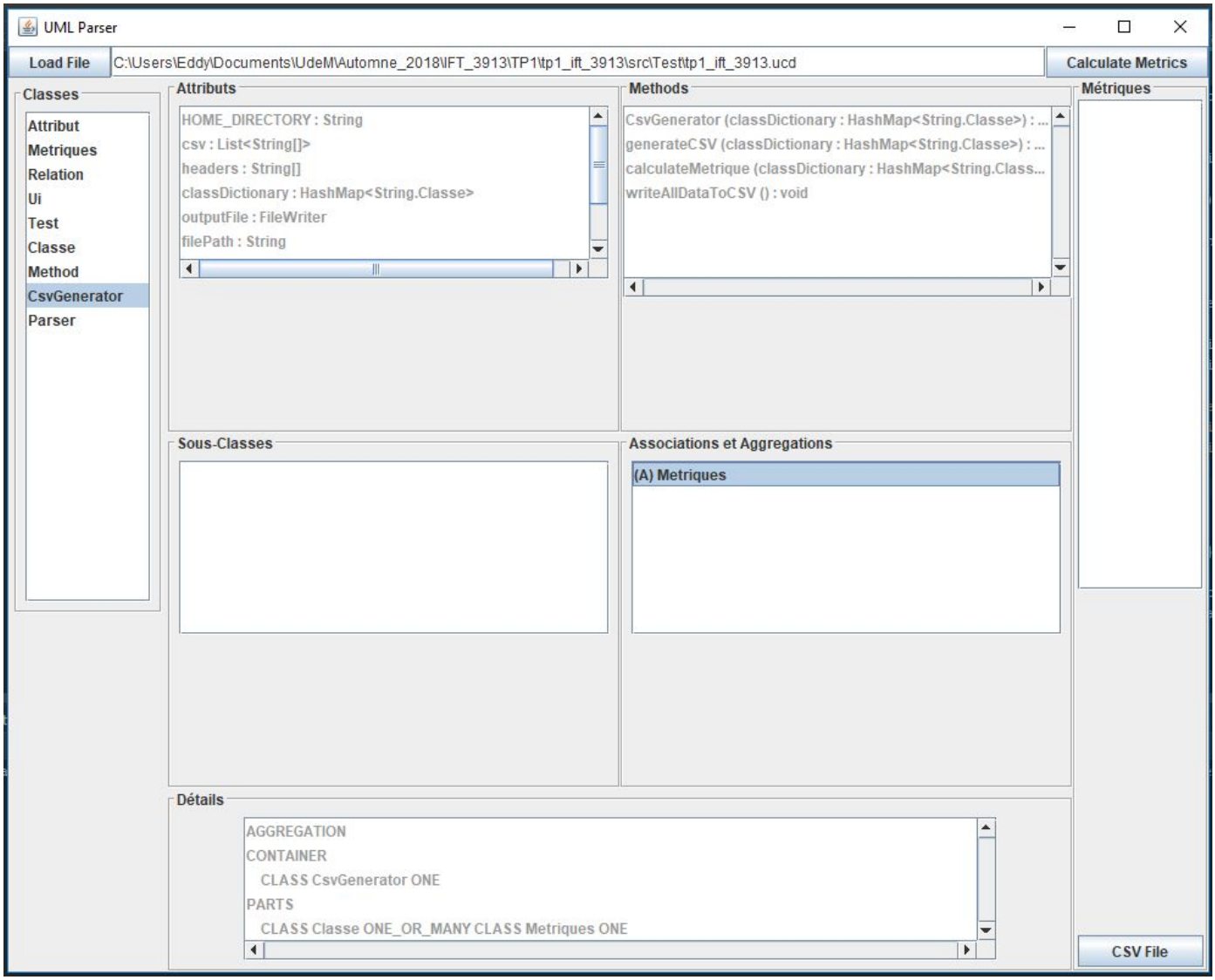



Choose UML diagram
Look In:
tp1_ift_3913
.git
.idea
out
src
Ligue.ucd
File Name:
Ligue.ucd
Files of Type:
UML Diagram
Open
Cancel

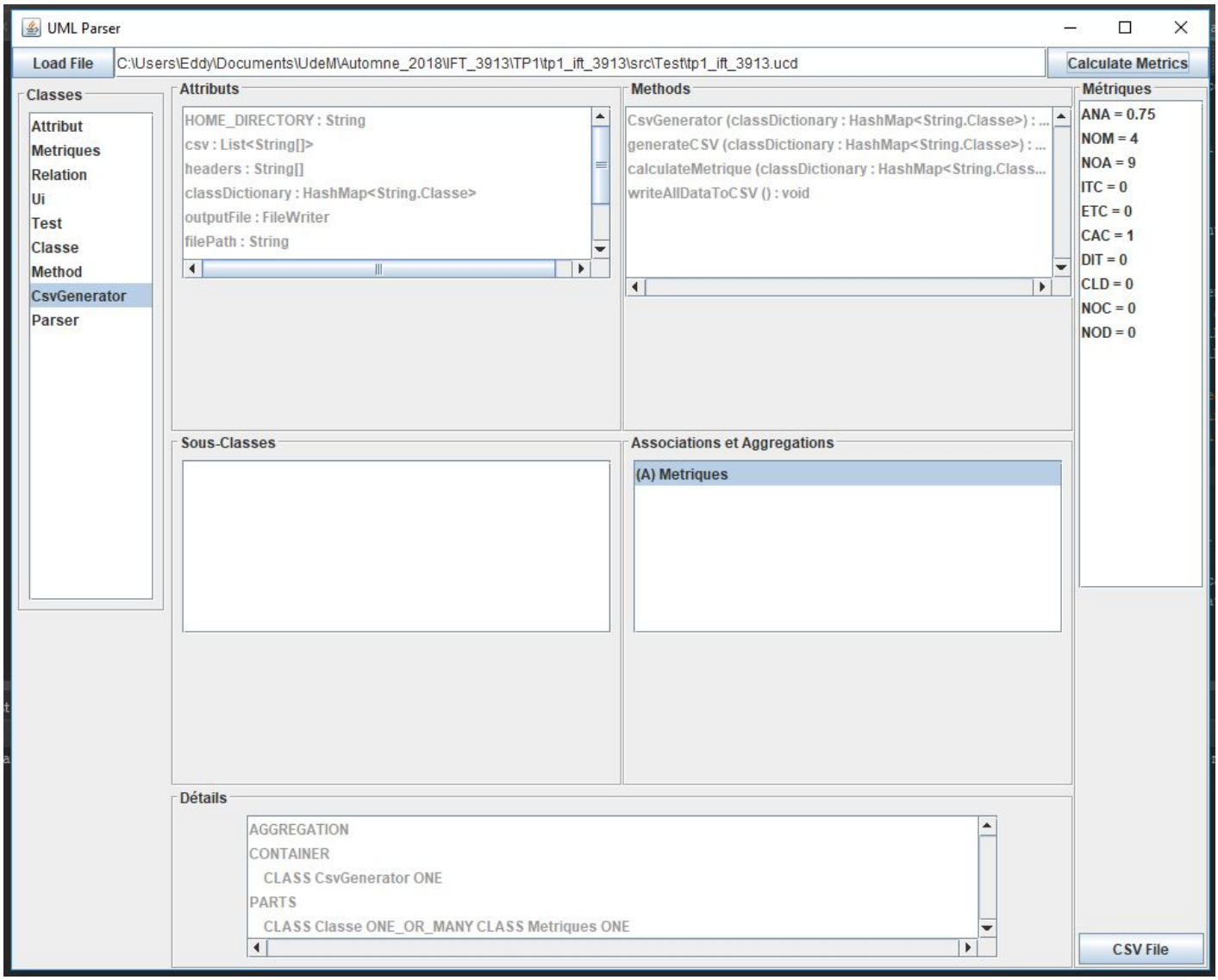
3. Une fois le fichier chargé dans l'interface utilisateur, vous serez en mesure de voir le chemin où vous avez choisi votre diagramme de classe. Nous avons décidé de laisser tout le chemin pour que l'utilisateur sache exactement d'où le fichier .ucd provient. Voilà l'interface qui vous fera la bienvenue.



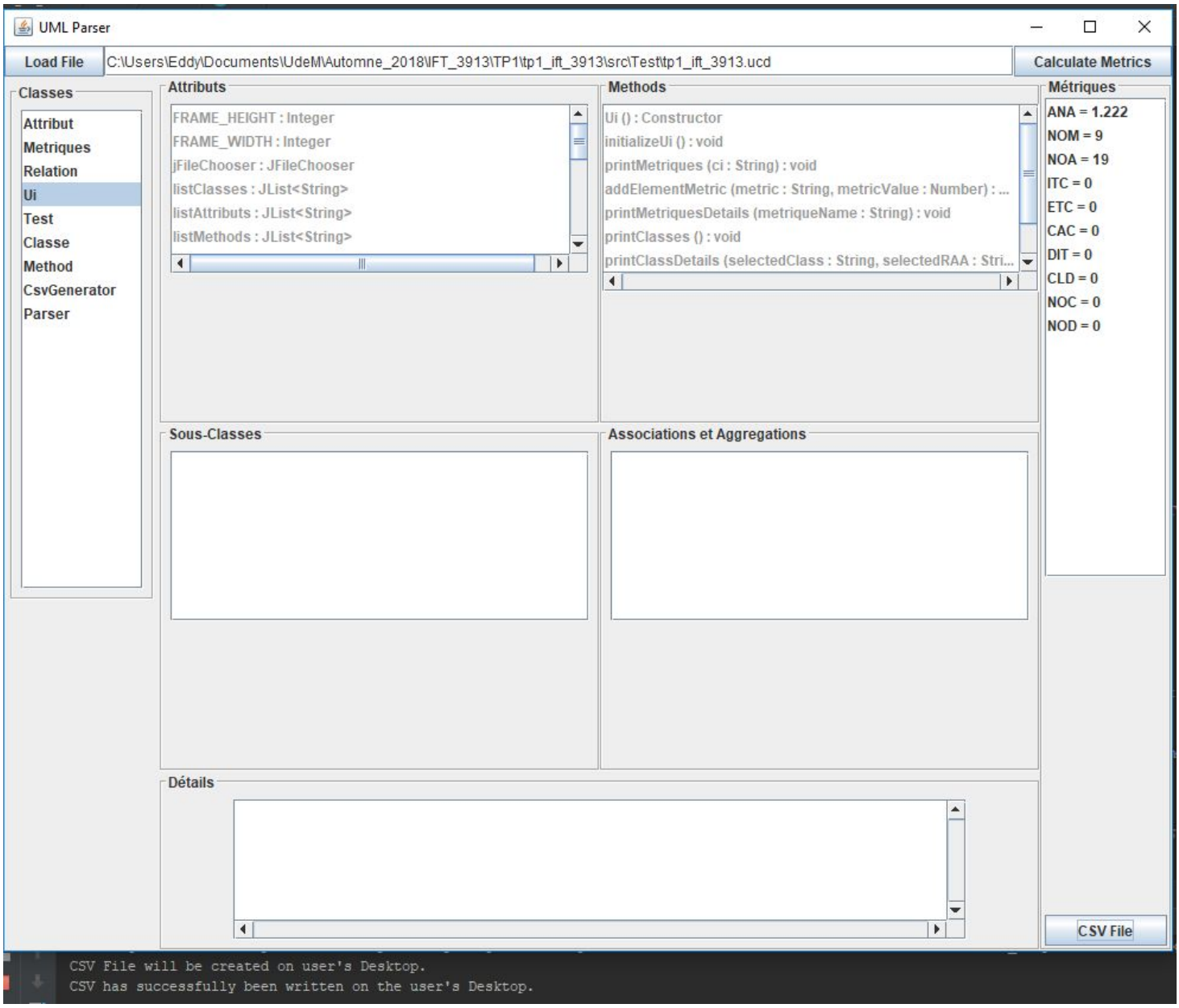
4. Par la suite, l'utilisateur peut s'amuser à cliquer sur une classe et voir de l'information apparaître devant lui. Il y a certain champ qui sont cliquable et d'autres qui ne le sont pas. Cela est allé à la discrétion et à la créativité des développeurs. Voilà l'interface que l'utilisateur verra.



5. Maintenant, avec les nouvelles fonctionnalités instaurées dans ce travail pratique. Vous n'avez qu'à appuyer sur une classe quelconque et appuyer sur le bouton *Calculate Metrics* et voilà. Les métriques seront calculées.



6. Pour finir, vous pouvez appuyer sur le boutons CSV File et un fichier CSV avec les métriques seront produites. **Prenez note:** Le fichier CSV sera créé sur votre Desktop. Comme vous pouvez voir, nous avons décidé d'imprimer un message vous annonçant que le fichier CSV a été créé avec succès.



7. Voilà le fichier CSV résultant

	A	B	C	D	E	F	G	H	I	J	K
1	Class Name	ANA	NOM	NOA	ITC	ETC	CAC	DIT	CLD	NOC	NOD
2	Attribut	0.714286	7	2	0	1	3	0	0	0	0
3	Metriques	0.888889	18	3	11	0	2	0	0	0	0
4	Relation	0.5	6	2	0	1	1	0	0	0	0
5	Ui	1.222222	9	19	0	0	0	0	0	0	0
6	Test	2.409091	22	25	1	0	0	0	0	0	0
7	Classe	0.705882	17	6	3	12	5	0	0	0	0
8	Method	0.8	10	3	1	0	3	0	0	0	0
9	CsvGenerator	0.75	4	9	0	0	1	0	0	0	0
10	Parser	1.375	8	7	0	0	1	0	0	0	0
11											
12											
13											

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 partie à plein coeur.