

Analyse statique et dynamique

Rapport pour TP1_Partie 2

Module : HAI913IÉvolution et restructuration des logiciels

Professeur : *Abdelhak DJamal Seriali*

Realisé par : BOUMEZGANE Hajar - 22414267

Année universitaire : 2025-2026

Plan

Introduction Générale.....	3
Conception et architecture de la solution	3
Partie 1 – Manipulation de l’AST avec Eclipse JDT.....	5
Synthèse de la Partie 1	6
Partie 2 : – Calculs statistiques et graphe d’appel	8
1 . Calcul statistique pour une application OO.....	8
1.1 - Statistiques calculées	8
1.2 - Interface graphique	9
2 . Construction du graphe d’appel d’une application.....	11
2.1 - Graphe textuel	11
2.2 - Graphe visuel avec JGraphX	12
Diagramme de classe UML.....	13

Introduction Générale

Ce rapport présente le travail réalisé dans le cadre du TP d'évolution et restructuration des logiciels. L'objectif principal est de manipuler l'**AST** (Abstract Syntax Tree) à l'aide de la bibliothèque **Eclipse JDT**, afin d'analyser des programmes écrits en Java.

L'outil développé extrait automatiquement des informations structurelles et comportementales du code, de calculer plusieurs statistiques, et de générer un graphe d'appel illustrant les relations entre les méthodes.

Une interface graphique avec deux onglets : *Statistiques* et *Graphe d'appel*, offre une visualisation interactive des résultats.

Enfin, un projet métier indépendant sert de cible d'analyse pour valider le bon fonctionnement de l'application.

Le code source complet du projet ainsi que le fichier README sont disponibles sur GitHub : https://github.com/Hajar-BH/TP_AnalyseStatique

Lien vers la vidéo de démonstration :

https://drive.google.com/file/d/14LTwO_ZicyZ9BfizJCbKeEIFvILw6UIM/view?usp=sharing

Conception et architecture de la solution

L'architecture du projet s'appuie sur le **workflow** classique d'une analyse statique par AST, tel qu'enseigné durant le cours.

Le Parser construit l'AST à partir du code source, les Visiteurs en parcourent les nœuds pour extraire les données nécessaires, puis l'AST Processor applique les traitements et envoie les résultats à l'interface graphique.

Le projet est divisé en **deux** modules :

- **ProjetAST** : contient l'outil d'analyse et la logique de traitement.
- **ProjetMetier** : sert de cible d'analyse pour valider les fonctionnalités de l'application.

Les packages sont organisés selon leurs rôles : parser, visitors, processor, graph, et ui.

Cette conception modulaire respecte fidèlement le schéma du cours et garantit une séparation claire entre l'analyse, le calcul des statistiques et la visualisation des résultats.

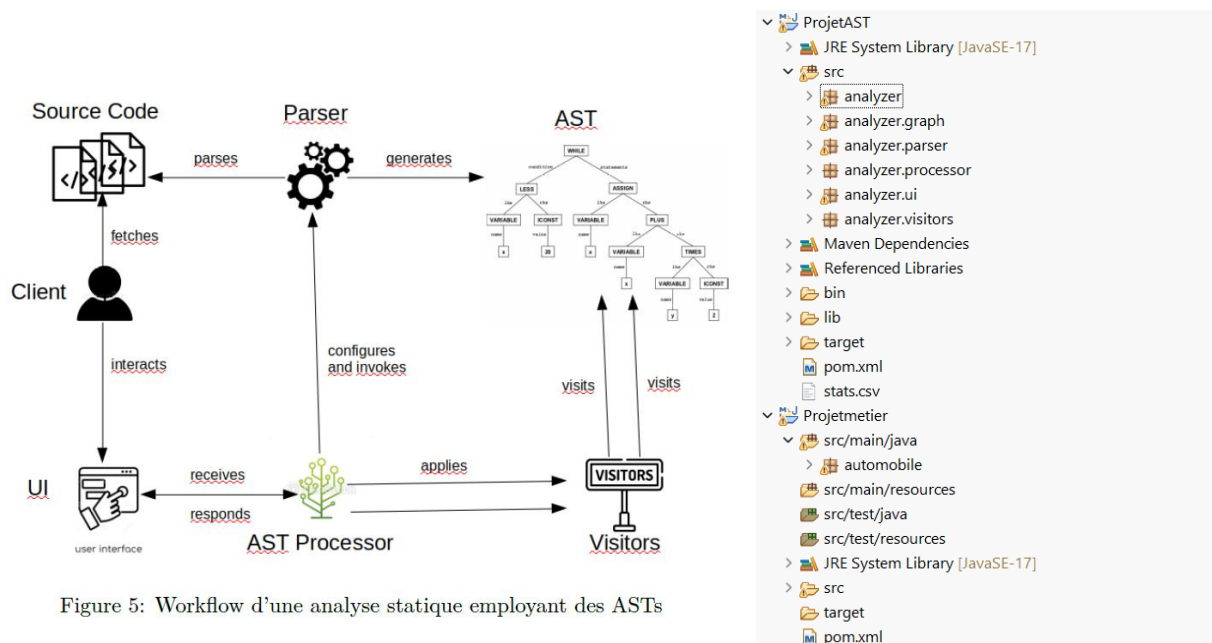


Figure 5: Workflow d'une analyse statique employant des ASTs

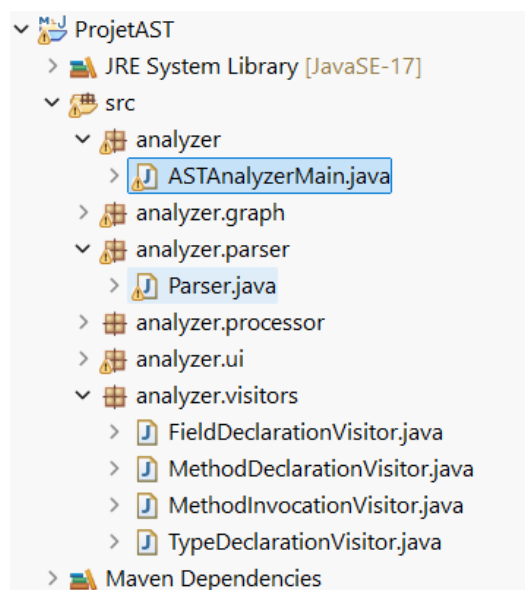
Partie 1 – Manipulation de l’AST avec Eclipse JDT

Avant d’aborder la **partie 2**, qui constitue le cœur de ce rendu, il est important de présenter la démarche suivie pour la Partie 1, qui constitue la base de toute l’analyse statique.

Cette étape nous a permis de comprendre la structure interne du code Java à travers l’AST (Abstract Syntax Tree) et d’apprendre à le manipuler grâce à la bibliothèque Eclipse JDT.

L’outil développé repose sur la classe `ASTParser`, qui transforme le code source en un arbre syntaxique, où chaque nœud représente un élément du langage : classe, méthode, attribut, etc.

À partir de cet arbre, différents visiteurs spécialisés parcourent le code pour extraire automatiquement les informations principales :



Pour illustrer le fonctionnement de ces visiteurs, voici un exemple de sortie console obtenu lors de l’analyse du projet *ProjetMetier* sur les classes *Driver* et *ElectricCar* :

```
Problems @ Javadoc Declaration Console X
ASTAnalyzerMain (1) [Java Application] C:\Users\User\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1
=== Analyse du fichier : Driver.java ===
Classe : Driver
  Attribut : name | Visibilité : private | Initialisation : null
Méthode : Driver | Retour : null
Méthode : drive | Retour : void
  Appel trouvé dans drive → println
  Appel trouvé dans drive → getModel
  Appel trouvé dans drive → accelerate
Méthode : testTruck | Retour : void
  Appel trouvé dans testTruck → println
  Appel trouvé dans testTruck → load
  Appel trouvé dans testTruck → brake

=== Analyse du fichier : ElectricCar.java ===
Classe : ElectricCar
  Hérite de : Car
  Attribut : batteryLevel | Visibilité : private | Initialisation : null
Méthode : ElectricCar | Retour : null
Méthode : chargeBattery | Retour : void
```

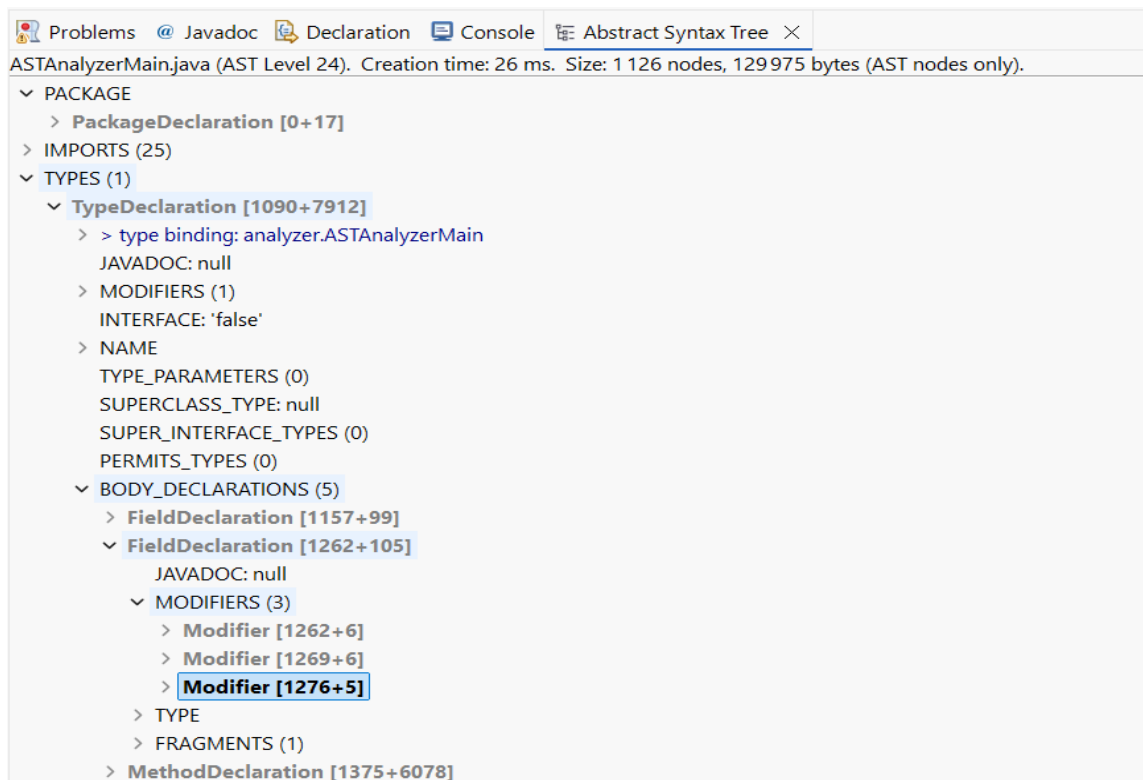
À noter que le projet *ProjetMetier* est utilisé ici à titre d'exemple par défaut. L'outil reste cependant générique et peut être appliqué à tout autre projet Java.

En complément, la figure suivante illustre la structure interne du code visualisée à l'aide de la vue *AST View* d'Eclipse.

Cette représentation montre la hiérarchie complète des éléments du fichier analysé :

déclarations de classes, d'attributs, de méthodes et leurs modificateurs.

Elle permet de mieux comprendre comment l'AST (Abstract Syntax Tree) décompose le code source en nœuds manipulables par les visiteurs de notre outil.



Synthèse de la Partie 1

Exercice 1 – Découverte de l'AST

Cet exercice m'a permis de comprendre comment l'API Eclipse JDT permet de générer et parcourir un arbre syntaxique abstrait (AST) à partir du code source Java.

J'ai appris à utiliser la classe `ASTParser` pour analyser la structure du code et à manipuler les différents nœuds représentant les classes, les méthodes et les attributs.

Exercice 2 – Extraction des informations

Dans cet exercice, j'ai développé plusieurs visiteurs spécialisés (classe, méthode, variable, attribut) afin d'extraire automatiquement les éléments essentiels du code : noms, types, portées et initialisations.

J'ai également conçu une classe principale `AnalyseurAST_main` qui coordonne le parser et les visiteurs pour afficher les résultats d'analyse de manière structurée.

Exercice 3 – Analyse des appels de méthodes

Cet exercice m'a permis d'ajouter une nouvelle phase d'analyse portant sur les appels de méthodes (`m()` et `super.m()`).

Grâce à cette étape, j'ai pu identifier les interactions entre les différentes méthodes d'un programme et poser les bases nécessaires à la construction du graphe d'appel qui sera développée dans la Partie 2.

En résumé, cette première partie m'a permis de mieux comprendre la structure interne du code Java et la manière dont les informations peuvent être extraites automatiquement à l'aide de l'AST.

Elle a posé les bases nécessaires pour la suite du projet, où ces données seront exploitées afin de générer des statistiques et de visualiser les dépendances entre les éléments du code.

Partie 2 : – Calculs statistiques et graphe d'appel

Après avoir exploré la manipulation de l'AST et l'extraction des informations structurelles dans la première partie, cette deuxième partie s'intéresse à l'exploitation de ces données pour produire des analyses statistiques et visualiser les relations d'appel entre les méthodes.

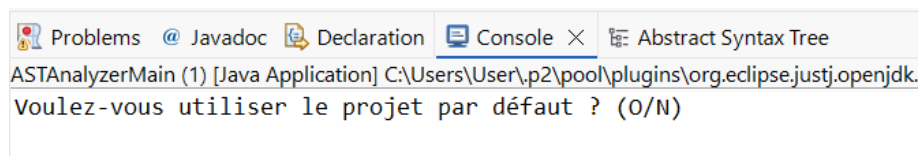
1 . Calcul statistique pour une application OO

Grâce au calcul statistique, il a été possible de mesurer automatiquement plusieurs caractéristiques du code source à partir des informations extraites via l'AST.

1.1- Statistiques calculées

La mise en œuvre de cette étape s'appuie sur un outil développé pour analyser automatiquement le code source d'un projet Java.

L'analyse peut être réalisée sur n'importe quel projet Java. Lors de l'exécution, **une question s'affiche dans la console** :



Si l'utilisateur saisit **O**, l'analyse se fera sur le projet *ProjetMetier*, défini par la constante *defaultProjectPath*.

Dans le cas contraire **N**, il est invité à saisir manuellement le chemin complet du projet qu'il souhaite analyser.

Pour centraliser les données collectées par les visiteurs de la Partie 1, j'ai implémenté la classe *ApplicationStatistics*, qui parcourt toutes les classes et méthodes détectées afin d'enregistrer leurs attributs, signatures, nombres de lignes et paramètres.

Elle calcule ensuite automatiquement les 13 mesures demandées dans le sujet, telles que : le nombre total de classes, de méthodes, de packages, les moyennes par classe ou par méthode, et l'identification des classes les plus complexes.

La structure et logique de la classe *ApplicationStatistics*

Cette classe repose sur trois étapes principales :

1. Collecte des données via les méthodes *ajouterClasse*, *ajouterMethode* et *ajouterPackage*.
2. Agrégation et calculs statistiques avec des moyennes et recherches de valeurs maximales.
3. Classements pour identifier les classes ou méthodes dominantes (top 10 %).

Exemple :

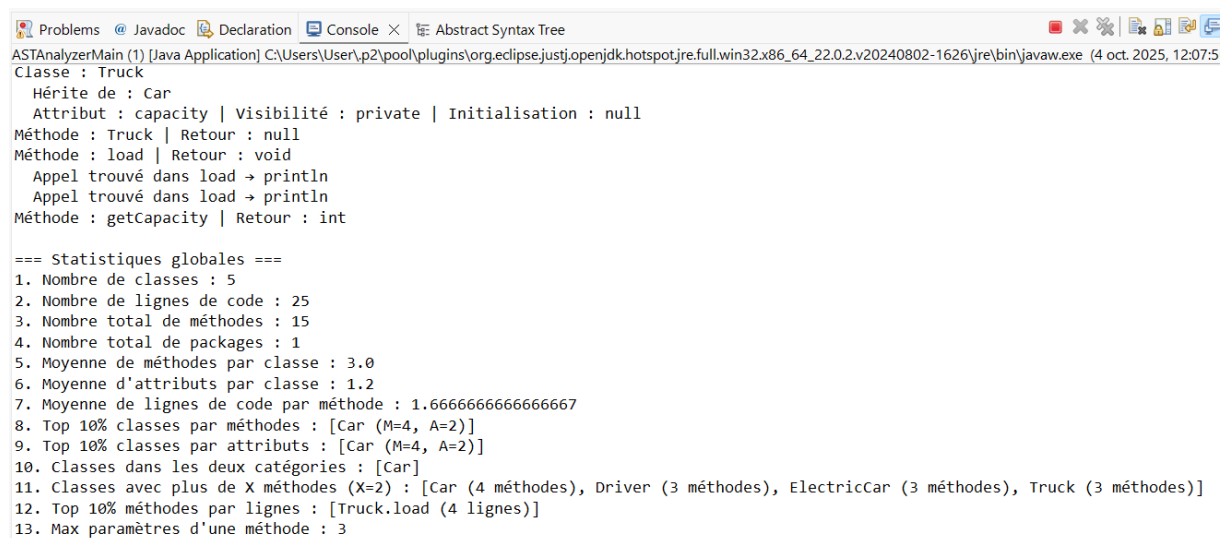
- la méthode *ajouterMethode* analyse le corps d'une méthode en comptant les lignes de code non vides ni commentées ; elle ajoute ensuite le nombre de paramètres et met à jour le total global.
- La méthode *ajouterClasse*, quant à elle, incrémente le compteur de classes et enregistre le nombre d'attributs et de méthodes de chacune.

Les fonctions *moyenneMethodesParClasse*, *moyenneLignesParMethode* ou encore *maxParametres* utilisent les flux Java (*stream()*) pour calculer efficacement les statistiques. Enfin, les méthodes *top10PourcentClassesParMethodes* et *top10PourcentClassesParAttributs* trient les résultats pour extraire les 10 % les plus significatifs.

À noter que pour la question 11, j'ai fixé la valeur de **X = 2** dans la classe *ASTAnalyzerMain*, afin d'identifier les classes possédant plus de deux méthodes.

Cette valeur peut être facilement ajustée dans l'appel à la méthode *classesAvecPlusDeXMethodes(int X)*, selon la taille ou la complexité du projet analysé.

Une fois l'analyse terminée, les résultats sont affichés dans la console, comme illustré ci-dessous.



```

Problems  Javadoc  Declaration  Console  Abstract Syntax Tree
ASTAnalyzerMain (1) [Java Application] C:\Users\User\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1626\jre\bin\javaw.exe (4 oct. 2025, 12:07:5
Classe : Truck
Hérite de : Car
Attribut : capacity | Visibilité : private | Initialisation : null
Méthode : Truck | Retour : null
Méthode : load | Retour : void
Appel trouvé dans load → println
Appel trouvé dans load → println
Méthode : getCapacity | Retour : int

=== Statistiques globales ===
1. Nombre de classes : 5
2. Nombre de lignes de code : 25
3. Nombre total de méthodes : 15
4. Nombre total de packages : 1
5. Moyenne de méthodes par classe : 3.0
6. Moyenne d'attributs par classe : 1.2
7. Moyenne de lignes de code par méthode : 1.6666666666666667
8. Top 10% classes par méthodes : [Car (M=4, A=2)]
9. Top 10% classes par attributs : [Car (M=4, A=2)]
10. Classes dans les deux catégories : [Car]
11. Classes avec plus de X méthodes (X=2) : [Car (4 méthodes), Driver (3 méthodes), ElectricCar (3 méthodes), Truck (3 méthodes)]
12. Top 10% méthodes par lignes : [Truck.load (4 lignes)]
13. Max paramètres d'une méthode : 3
  
```

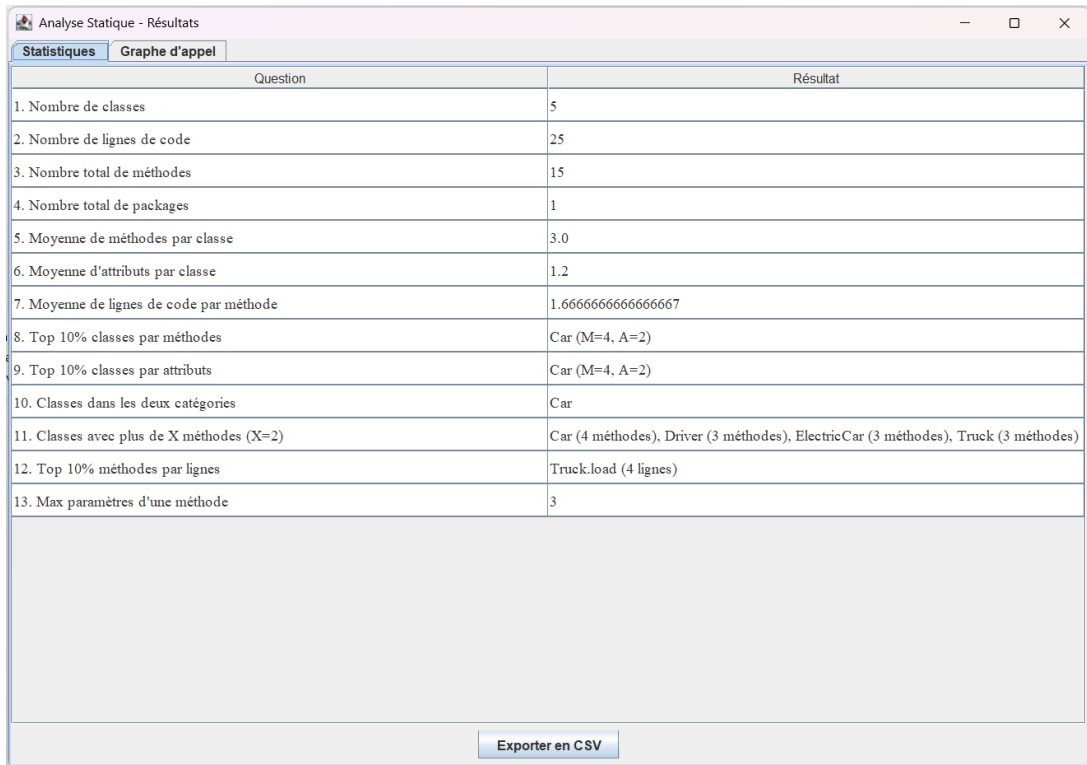
1.2- Interface graphique

Pour rendre l'analyse plus visuelle, j'ai développé une interface graphique avec **Swing**.

Cette interface, représentée dans la classe *MainWindow*, se compose de deux onglets principaux :

- Onglet Statistiques : géré par la classe *StatisticsWindow*, il affiche les 13 indicateurs calculés sous forme de tableau (**JTable**) clair et structuré.
Un bouton d'exportation **CSV** permet également de sauvegarder les résultats pour un usage ultérieur ou une analyse externe.
- Onglet Graphe d'appel : généré par les classes *CallGraph* et *CallGraphWindow*, cet onglet illustre visuellement les relations entre les méthodes de l'application.
C'est ce que nous allons détailler dans l'exercice qui suit.

La figure suivante illustre l'interface graphique de l'onglet **Statistiques**, affichant les différents indicateurs calculés à partir de l'analyse du projet.

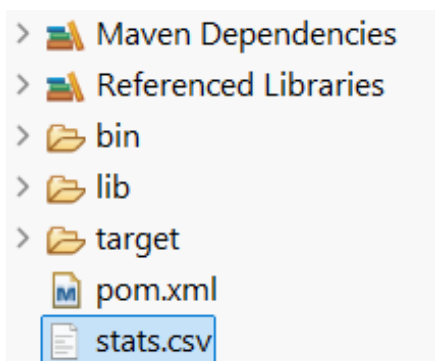
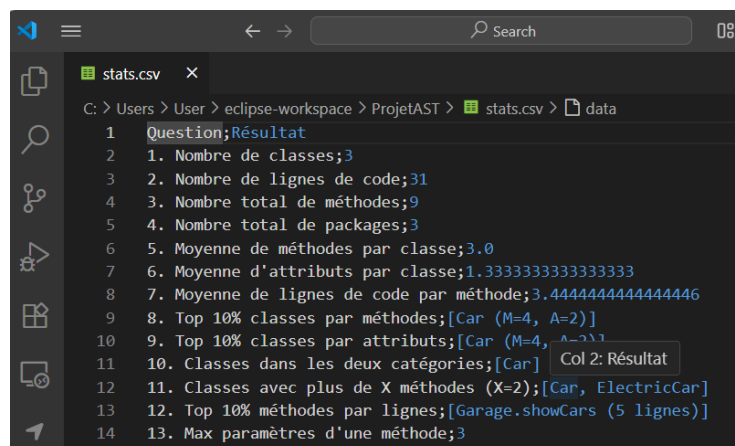


Question	Résultat
1. Nombre de classes	5
2. Nombre de lignes de code	25
3. Nombre total de méthodes	15
4. Nombre total de packages	1
5. Moyenne de méthodes par classe	3.0
6. Moyenne d'attributs par classe	1.2
7. Moyenne de lignes de code par méthode	1.6666666666666667
8. Top 10% classes par méthodes	Car (M=4, A=2)
9. Top 10% classes par attributs	Car (M=4, A=2)
10. Classes dans les deux catégories	Car
11. Classes avec plus de X méthodes (X=2)	Car (4 méthodes), Driver (3 méthodes), ElectricCar (3 méthodes), Truck (3 méthodes)
12. Top 10% méthodes par lignes	Truck.load (4 lignes)
13. Max paramètres d'une méthode	3

Exporter en CSV

Grâce à cette interface, l'utilisateur peut à la fois observer les statistiques globales et consulter rapidement les moyennes, les totaux, ainsi que les classes et méthodes dominantes. Elle constitue également une première visualisation claire, simple et intuitive des résultats produits par l'analyse statique.

De plus, j'ai ajouté la possibilité d'exporter les résultats au format CSV. En cliquant sur le bouton Exporter en CSV, un fichier nommé **stats.csv** est automatiquement généré à la racine du projet.

```

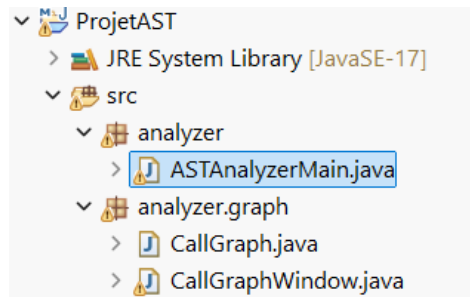
1 Question;Résultat
2 1. Nombre de classes;3
3 2. Nombre de lignes de code;31
4 3. Nombre total de méthodes;9
5 4. Nombre total de packages;3
6 5. Moyenne de méthodes par classe;3.0
7 6. Moyenne d'attributs par classe;1.3333333333333333
8 7. Moyenne de lignes de code par méthode;3.4444444444444446
9 8. Top 10% classes par méthodes;[Car (M=4, A=2)]
10 9. Top 10% classes par attributs;[Car (M=4, A=2)]
11 10. Classes dans les deux catégories;[Car]
12 11. Classes avec plus de X méthodes (X=2);[Car, ElectricCar]
13 12. Top 10% méthodes par lignes;[Garage.showCars (5 lignes)]
14 13. Max paramètres d'une méthode;3

```

2 . Construction du graphe d'appel d'une application

Cette dernière partie m'a permis de mettre en place un graphe d'appel représentant les relations entre les méthodes de l'application.

L'objectif est de visualiser les dépendances internes au code, c'est-à-dire quelles méthodes en appellent d'autres, afin de mieux comprendre la structure et les interactions de l'application analysée.



2.1 - Graphe textuel

La première étape consistait à générer une représentation textuelle du graphe d'appel à l'aide de la classe CallGraph.

Cette classe gère la structure logique du graphe : chaque méthode est enregistrée comme nœud, et les appels entre méthodes sont représentés sous forme d'arêtes dirigées.

Les informations sont stockées dans une structure `Map<String, Set<String>>`, où chaque clé correspond à une méthode appelante et sa valeur à l'ensemble des méthodes appelées.

La méthode *ajouterAppel()* enregistre les appels détectés par le visiteur *MethodInvocationVisitor*, tandis que la méthode *toString()* permet d'obtenir une représentation textuelle complète du graphe, affichée directement dans la console.

Exemple de sortie console :

```
Problems  @ Javadoc  Declaration  Console  Abstract Syntax Tree
ASTAnalyzerMain (1) [Java Application] C:\Users\User\p2\pool\plugins\org.eclipse.justj.

===== Graphe d'appel =====
testTruck → [brake, println, load]
load → [println]
addCar → [add]
showCars → [println, getModel]
drive → [println, getModel, accelerate]
```

On peut observer que testTruck appelle brake, println et load, tandis que showCars appelle println et getModel. Ce premier affichage textuel m'a permis de vérifier la cohérence du graphe avant de passer à la visualisation graphique.

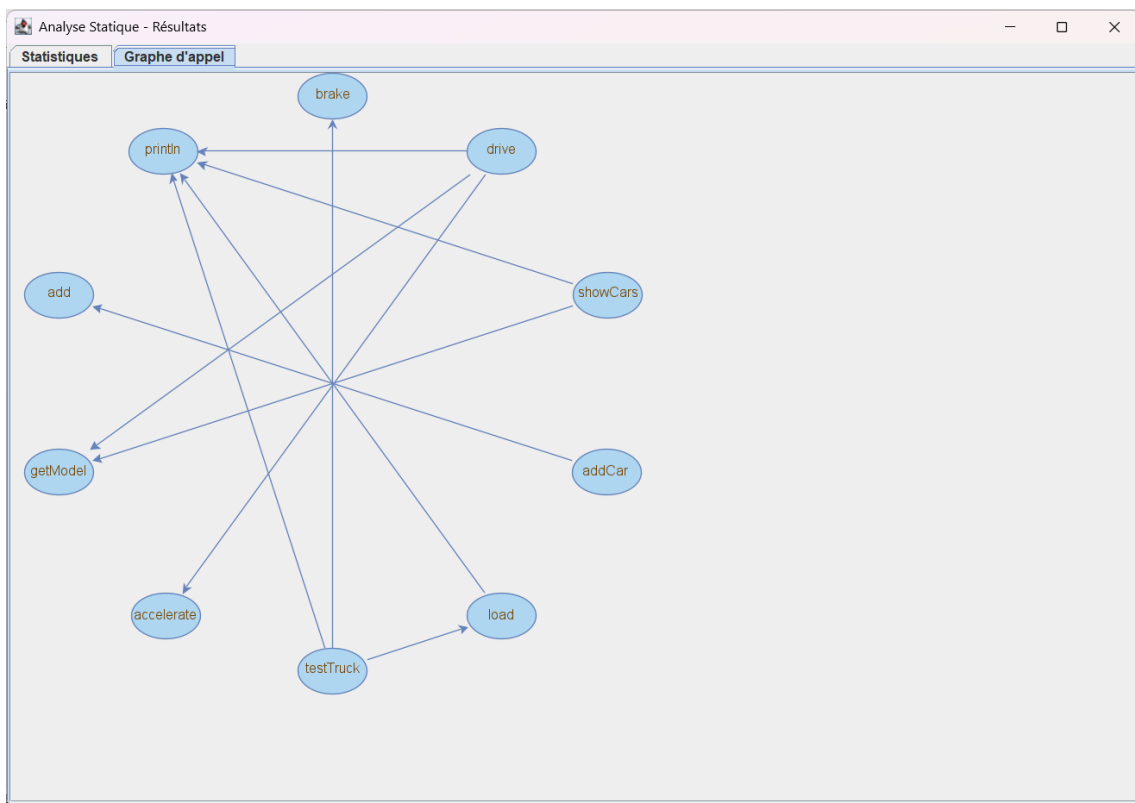
2.2- Graphe visuel avec JGraphX

Après le graphe textuel, j'ai développé une visualisation graphique (**GUI**) à l'aide de la bibliothèque **JGraphX**.

Cette partie est gérée par la classe *CallGraphWindow*, qui affiche le graphe d'appel dans une interface Swing.

Chaque méthode est affichée sous forme d'un nœud elliptique, et les appels entre méthodes sont reliés par des arêtes orientées, disposées automatiquement via un layout circulaire (*mxCircleLayout*).

Ce choix de disposition permet d'obtenir un graphe équilibré et lisible, où toutes les relations d'appel sont clairement centrées dans la fenêtre, comme le montre la figure suivante :



Cette deuxième partie m'a permis d'aller plus loin dans l'analyse statique en intégrant à la fois des mesures quantitatives et une visualisation graphique intuitive, offrant une meilleure compréhension des dépendances entre les éléments du code.

Diagramme de classe UML

Enfin, afin de mettre en pratique les bases acquises l'année dernière dans le cours de conception, j'ai réalisé un diagramme UML illustrant la structure globale de l'outil d'analyse statique développé.

Ce diagramme met en évidence les principaux packages (*parser*, *processor*, *visitors*, *graph*, *ui*) ainsi que leurs relations, et montre l'interaction entre l'application d'analyse (*analyzer*) et le projet métier analysé (*ProjetMetier*).

