

# TP1 — **Partie 2** : Métriques & Graphe d'appel (JDT)

Réalisé par : **Ouezzani Rahma**

**Projet** : Partie2TPEvo (Eclipse + Maven)

**Environnement** : Ubuntu 22.04 — JDK 17 — Eclipse — JDT Core

---

## SOMMAIRE

### Table des matières

<b>1</b>	<b>Objectifs</b>	<b>1</b>
<b>2</b>	<b>Organisation du projet</b>	<b>2</b>
<b>3</b>	<b>Lancement</b>	<b>3</b>
<b>4</b>	<b>Exercice 1 — Métriques (Q1.1) + GUI</b>	<b>3</b>
<b>5</b>	<b>Exercice 2 — Graphe d'appel (Q2.1) + GUI</b>	<b>6</b>
<b>6</b>	<b>Résultats obtenus</b>	<b>8</b>
<b>7</b>	<b>Reproductibilité &amp; commandes</b>	<b>9</b>
<b>8</b>	<b>Limites, problèmes rencontrés &amp; pistes</b>	<b>9</b>
<b>9</b>	<b>Annexes</b>	<b>9</b>
<b>10</b>	<b>Liens du projet ( Github et Google Drive )</b>	<b>10</b>
<b>11</b>	<b>Projet métier complémentaire — company</b>	<b>10</b>

## Bonus réalisés

- **Q1.2 — GUI Métriques (réalisé)** : fenêtre Swing avec onglets *Résumé* et *Classes*.
- **Q2.2 — GUI Graphe d'appel (réalisé)** : visualisation graphique du graphe d'appel.
- **Exports reproductibles** : CSV des métriques, DOT du graphe (+ PNG via GraphViz).
- **Arguments CLI** : `-src` (racine des sources), `-gui` (ouvre la fenêtre), seuil **X** pour Q1.1(11).

*Note* : le graphe d'appel est **statique** (pas de dispatch dynamique ni réflexion), ce qui suffit pour l'exemple étudié.

## 1 Objectifs

- Parcourir du code Java avec **JDT** pour calculer des **métriques** : #classes, #méthodes, LOC, moyennes, tops (%).
- Construire un **graphe d'appel** statique (nœuds = méthodes, arêtes = appels).
- Produire des sorties **reproductibles** (console, CSV, DOT) et une **GUI** simple pour visualiser.

### Ce que j'ai appris

- JDT sait résoudre les *bindings* (types/méthodes), ce qui permet d'obtenir des noms qualifiés et donc un graphe propre.
- Compter des LOC fiables demande d'enlever les commentaires et les lignes vides avant de compter.

## 2 Organisation du projet

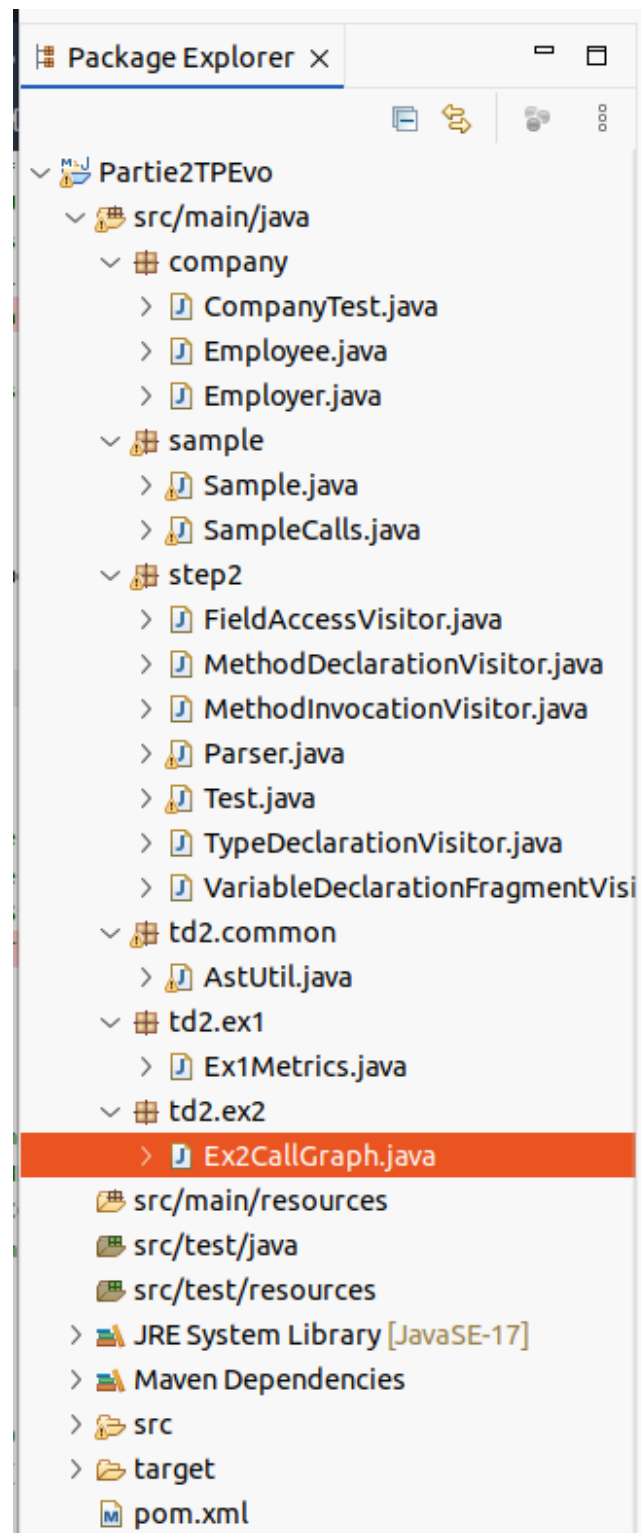


FIGURE 1 – Structure du projet (Eclipse — Package Explorer).

**Choix perso** j'ai séparé `common/` (utilitaires partagés) de `ex1/` et `ex2/` pour que chaque exercice reste clair et modulaire.

### 3 Lancement

Les deux programmes acceptent `--src=<dossier>` (racine des sources, défaut `src/main/java/sample`) et `--gui` pour ouvrir la fenêtre. Pour Q1.1(11), j'utilise **X** comme seuil (ex. 4).

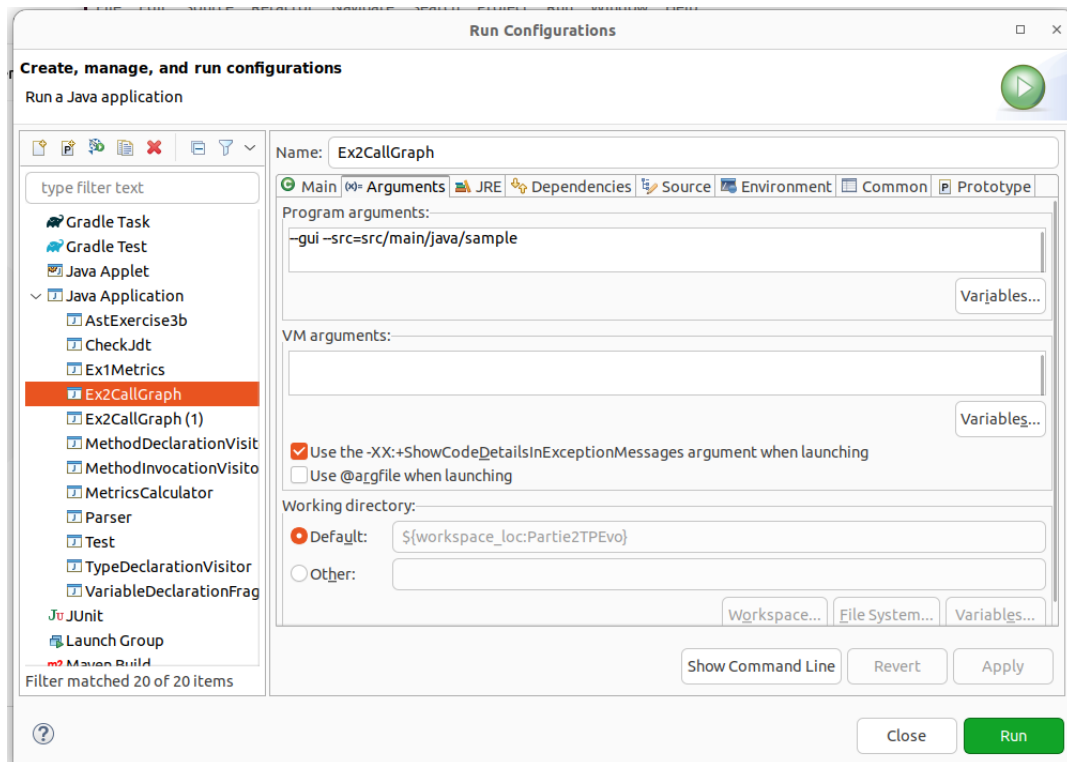


FIGURE 2 – Run Configs → *Arguments* (exemples montrés).

- **Ex1Metrics** : `--gui --src=src/main/java/sample 4`
- **Ex2CallGraph** : `--gui --src=src/main/java/sample`

### 4 Exercice 1 — Métriques (Q1.1) + GUI

#### Comment j'ai procédé

1. Je construis l'AST JDT (avec bindings) à partir des sources (`AstUtil.parse`).
2. Je visite les `TypeDeclaration` et `MethodDeclaration`.
3. Je calcule :
  - **LOC global** : je retire `//...` et `/*...*/` puis je compte les lignes non vides.
  - **LOC/méthode** : je prends le *corps* uniquement (pas la signature).
  - **Tops 10%** : j'utilise  $k = \max(1, \lfloor 0,1 \times n \rfloor)$ .

#### Mini-extrait de code (idée générale)

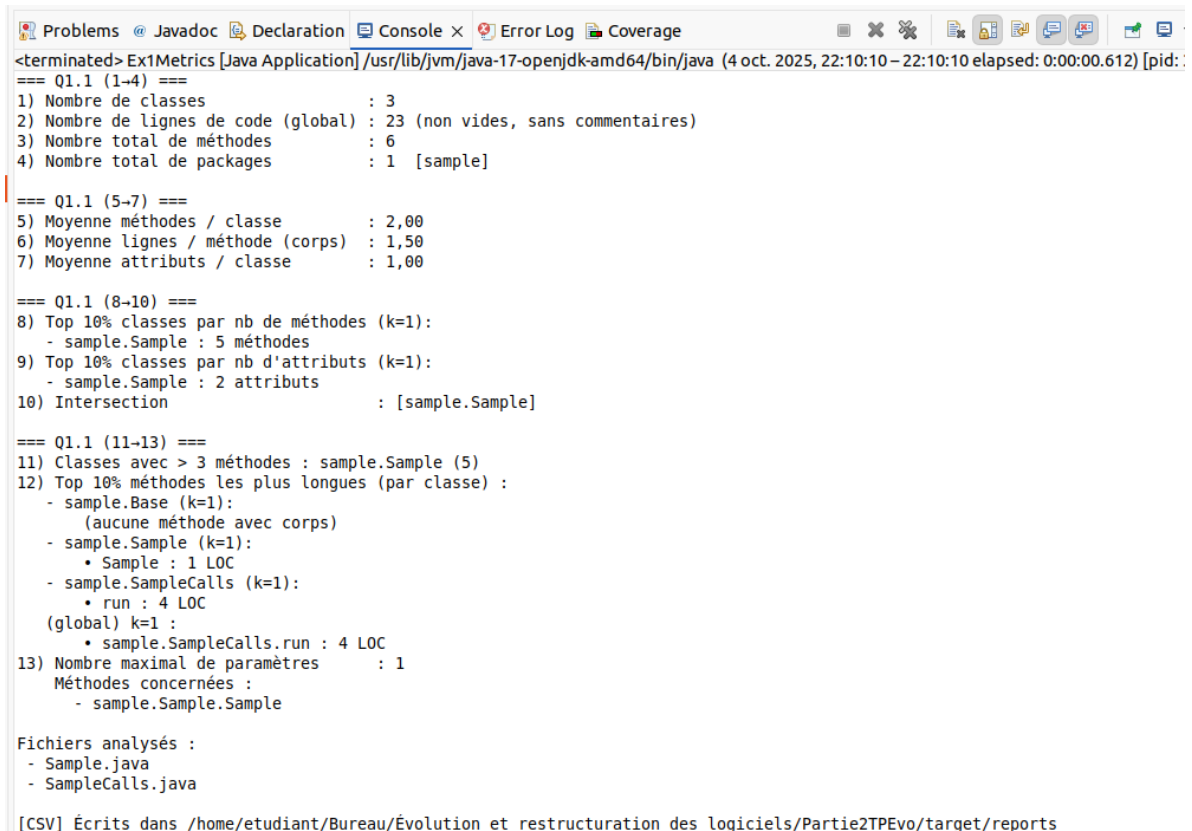
```
1 // Q1.1      parcours et agrégation
2 CompilationUnit cu = AstUtil.parse(content.toCharArray(), file);
3 cu.accept(new ASTVisitor() {
4     @Override public boolean visit(TypeDeclaration td) {
5         String fqcn = AstUtil.fqn(cu, td.getName().getIdentifier());
6         classes.computeIfAbsent(fqcn, ClassInfo::new);
7         return true;
8     }
9     @Override public boolean visit(MethodDeclaration md) {
```

```

10     String owner = AstUtil.enclosingTypeFqn(cu, md);
11     int locBody = AstUtil.countLOCInBody(md); // lignes non vides du bloc
12     classes.get(owner).addMethod(new MethodInfo(owner, md.getName().
        toString(), md.parameters().size(), locBody));
13     return true;
14 }
15 });

```

## Ce que j'affiche



```

<terminated> Ex1Metrics [Java Application] /usr/lib/jvm/java-17-openjdk-amd64/bin/java (4 oct. 2025, 22:10:10 – 22:10:10 elapsed: 0:00:00.612) [pid:
=== Q1.1 (1-4) ===
1) Nombre de classes : 3
2) Nombre de lignes de code (global) : 23 (non vides, sans commentaires)
3) Nombre total de méthodes : 6
4) Nombre total de packages : 1 [sample]

=== Q1.1 (5-7) ===
5) Moyenne méthodes / classe : 2,00
6) Moyenne lignes / méthode (corps) : 1,50
7) Moyenne attributs / classe : 1,00

=== Q1.1 (8-10) ===
8) Top 10% classes par nb de méthodes (k=1):
- sample.Sample : 5 méthodes
9) Top 10% classes par nb d'attributs (k=1):
- sample.Sample : 2 attributs
10) Intersection : [sample.Sample]

=== Q1.1 (11-13) ===
11) Classes avec > 3 méthodes : sample.Sample (5)
12) Top 10% méthodes les plus longues (par classe) :
- sample.Base (k=1):
  (aucune méthode avec corps)
- sample.Sample (k=1):
  • Sample : 1 LOC
- sample.SampleCalls (k=1):
  • run : 4 LOC
(global) k=1 :
  • sample.SampleCalls.run : 4 LOC
13) Nombre maximal de paramètres : 1
    Méthodes concernées :
    - sample.Sample.Sample

Fichiers analysés :
- Sample.java
- SampleCalls.java

[CSV] Écrits dans /home/etudiant/Bureau/Évolution et restructuration des logiciels/Partie2TPEvo/target/reports

```

FIGURE 3 – Console — Q1.1 (1→13) avec mes chiffres.

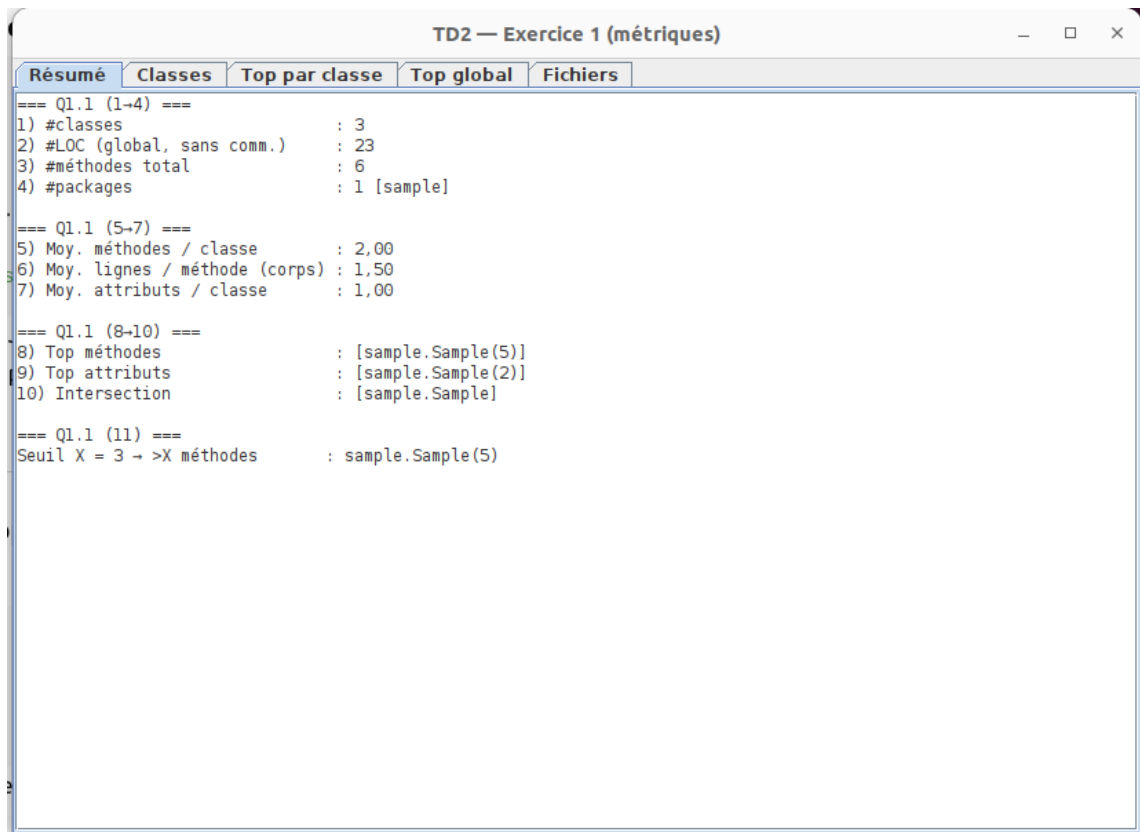


FIGURE 4 – GUI — onglet *Résumé* (bonus).

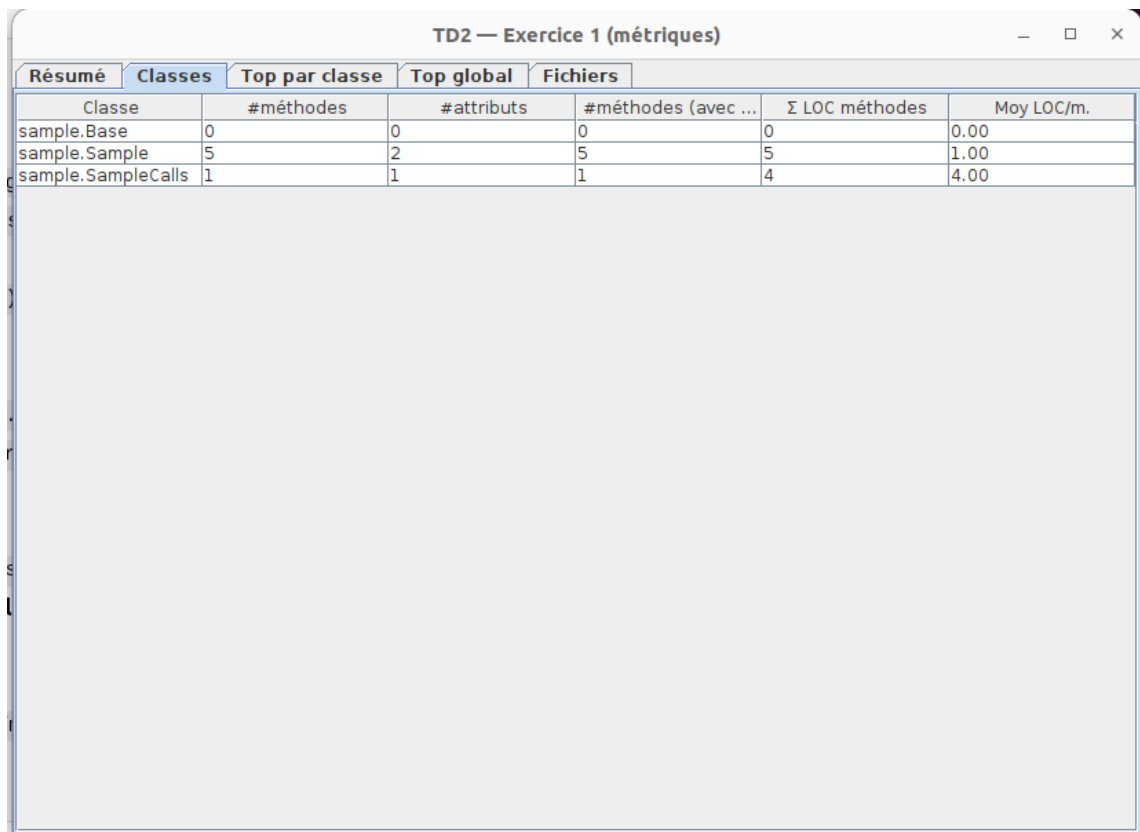


FIGURE 5 – GUI — onglet *Classes* (tableau récapitulatif).

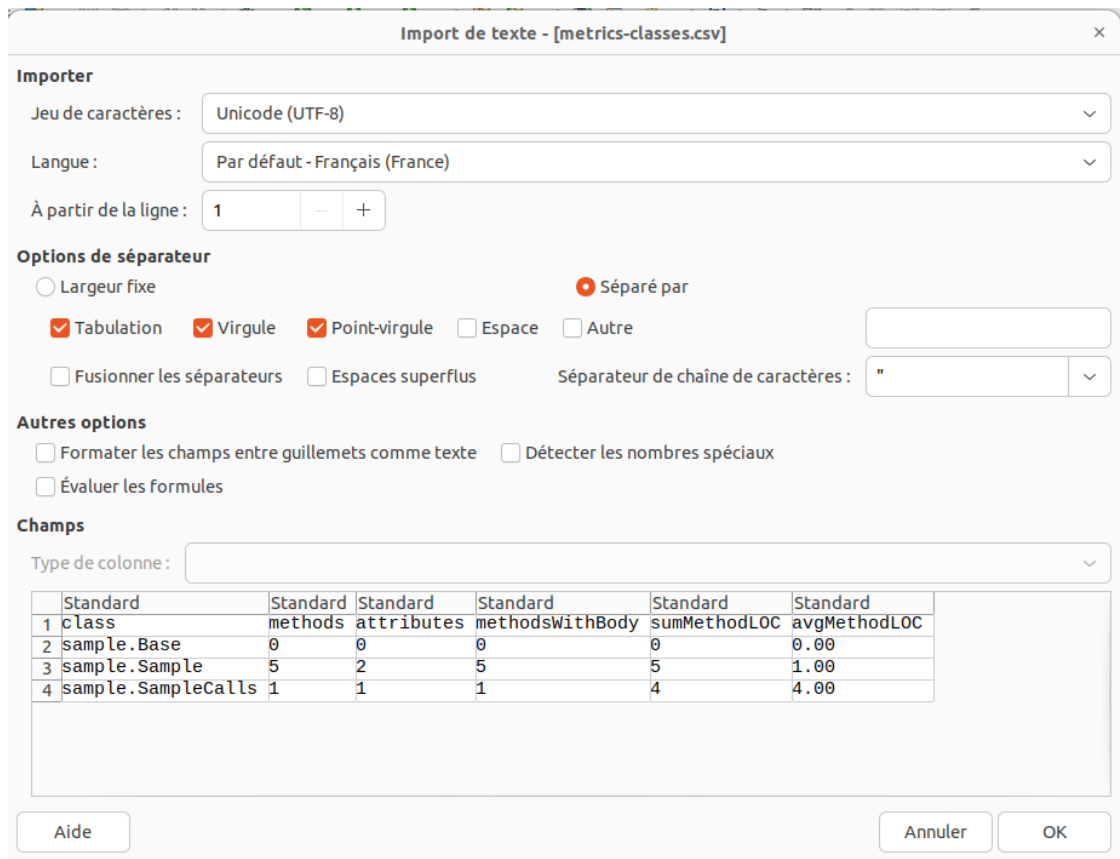


FIGURE 6 – Un CSV produit dans `target/reports`.

**Petite note** j'ai préféré stocker aussi les CSV pour pouvoir rejouer/partager les résultats sans relancer l'outil.

## 5 Exercice 2 — Graphe d'appel (Q2.1) + GUI

### Comment j'ai construit le graphe

- **Nœud** = `ClasseFQN.méthode` (ou `<init>` pour un constructeur).
- **Arête** = `caller → callee`.

Pour chaque `MethodDeclaration`, je visite :

- `MethodInvocation (obj.m())` ⇒ je résous le binding pour connaître le type déclarant.
- `SuperMethodInvocation (super.m())`.
- `ClassInstanceCreation (new T(...))` ⇒ j'ajoute une arête vers `T.<init>`.

### Mini-extraits de code

```

1 // Dans chaque méthode (caller), je balaie le corps :
2 @Override public boolean visit(MethodInvocation inv) {
3     IMethodBinding mb = inv.resolveMethodBinding();
4     String recv = (mb != null && mb.getDeclaringClass() != null)
5         ? mb.getDeclaringClass().getQualifiedName() : "<inconnu>";
6     graph.addEdge(currentMethodFqn, recv + "." + inv.getName().getIdentifier());
7     return true;
8 }
9 @Override public boolean visit(ClassInstanceCreation cic) {
10     ITypeBinding tb = cic.resolveTypeBinding();

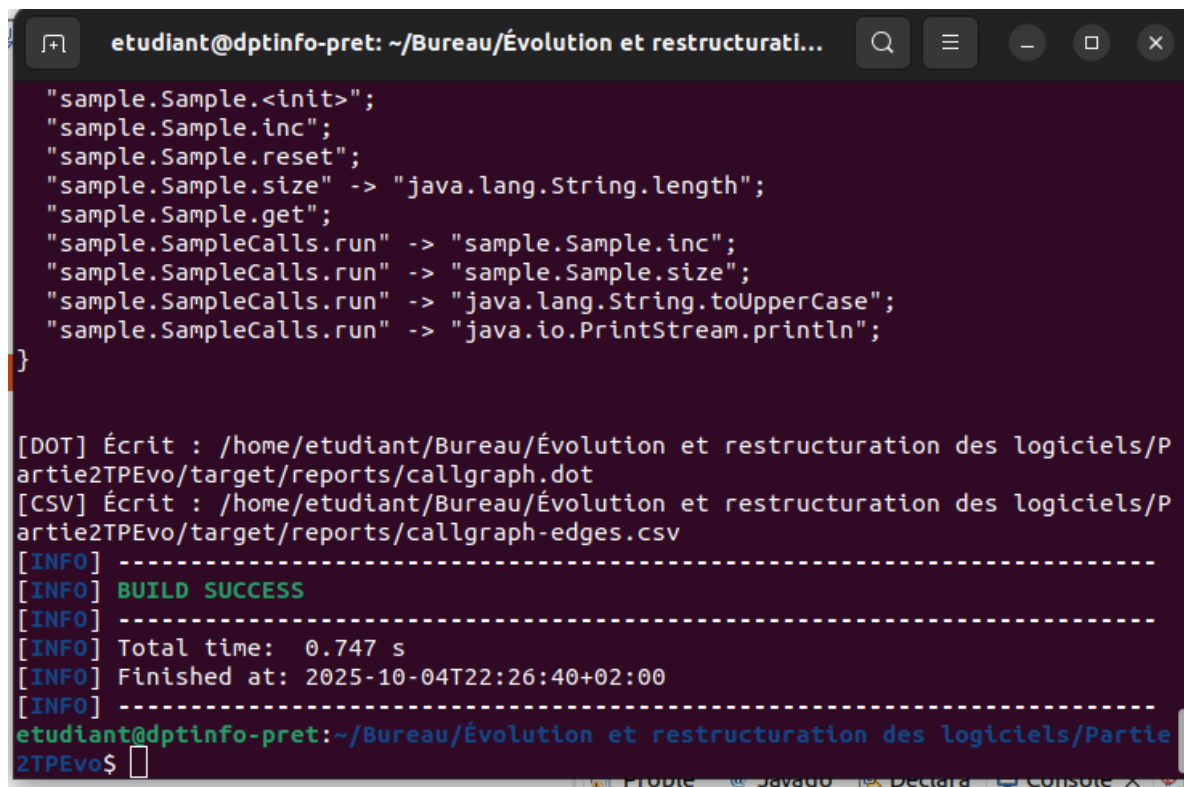
```

```

11 String ctor = (tb != null) ? tb.getQualifiedName()+".<init>" : "<inconnu
    >.<init>";
12 graph.addEdge(currentMethodFqn, ctor);
13 return true;
14 }

```

## DOT, CSV et GUI



```

etudiant@dptinfo-pret: ~/Bureau/Évolution et restructurati...
"sample.Sample.<init>";
"sample.Sample.inc";
"sample.Sample.reset";
"sample.Sample.size" -> "java.lang.String.length";
"sample.Sample.get";
"sample.SampleCalls.run" -> "sample.Sample.inc";
"sample.SampleCalls.run" -> "sample.Sample.size";
"sample.SampleCalls.run" -> "java.lang.String.toUpperCase";
"sample.SampleCalls.run" -> "java.io.PrintStream.println";
}

[DOT] Écrit : /home/etudiant/Bureau/Évolution et restructuration des logiciels/P
artie2TPEvo/target/reports/callgraph.dot
[CSV] Écrit : /home/etudiant/Bureau/Évolution et restructuration des logiciels/P
artie2TPEvo/target/reports/callgraph-edges.csv
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  0.747 s
[INFO] Finished at: 2025-10-04T22:26:40+02:00
[INFO] -----
etudiant@dptinfo-pret:~/Bureau/Évolution et restructuration des logiciels/Partie
2TPEvo$

```

FIGURE 7 – Console — bloc DOT + fichier callgraph.dot écrit.



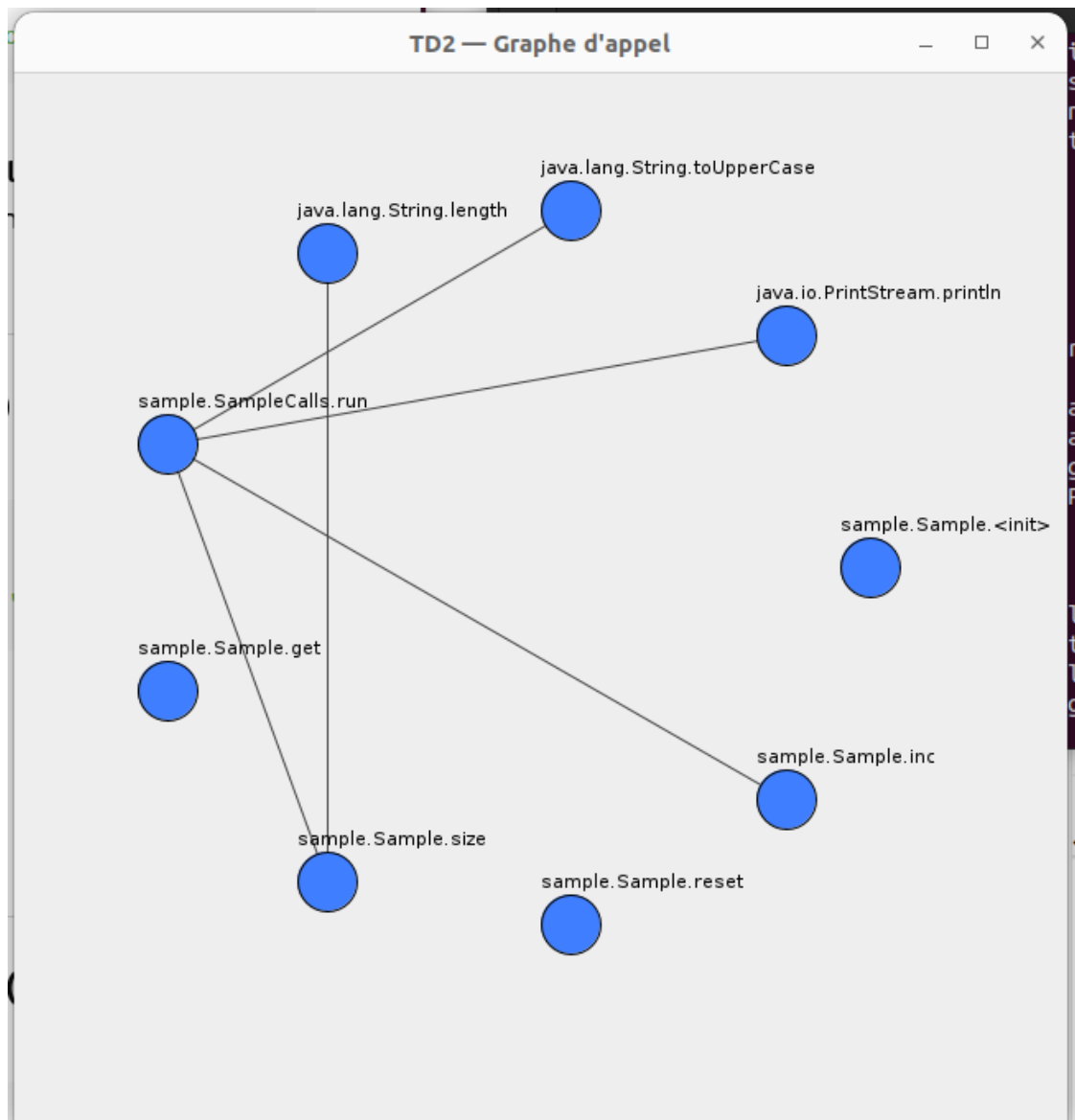


FIGURE 8 – GUI — graphe d'appel (bonus).

Astuce (optionnelle) pour générer une image :

```
dot -Tpng target/reports/callgraph.dot -o target/reports/callgraph.png
```

## 6 Résultats obtenus

### Métriques (sur sample/)

- #classes = **3** ; #méthodes = **6** ; #packages = **1** ; #LOC global = **23**.
- Moyennes : méthodes/classe = **2,00** ; LOC/méthode = **1,50** ; attributs/classe = **1,00**.
- Top 10% classes (méthodes) : **sample.Sample** (5) ; Top 10% attributs : **sample.Sample** (2) ; Intersection : [**sample.Sample**].
- (11) > 3 méthodes : **sample.Sample** (5). (12) Méthode la plus longue (global) : **sample.SampleCalls.run** = **4 LOC**. (13) Max #params = **1** (constructeur de **Sample**).

### Graphe d'appel (arêtes notables)

**sample.Sample.size** → **java.lang.String.length**

```
sample.SampleCalls.run → sample.Sample.inc
sample.SampleCalls.run → sample.Sample.size
sample.SampleCalls.run → java.lang.String.toUpperCase
sample.SampleCalls.run → java.io.PrintStream.println
```

## 7 Reproductibilité & commandes

- **Ex1Metrics** : `--gui --src=src/main/java/sample 4`
- **Ex2CallGraph** : `--gui --src=src/main/java/sample`
- **DOT→PNG (optionnel)** : `dot -Tpng target/reports/callgraph.dot -o target/reports/callgrap`

## 8 Limites, problèmes rencontrés & pistes

### Ce qui m'a bloquée un moment

- Conflits d'imports (`java.awt.Dimension` vs. `JDT`) : j'ai gardé les imports **JDT** et évité les noms ambigus.
- `#LOC` : j'ai dû bien filtrer `//...` et `/*...*/` avant de compter.

## 9 Annexes

### Parsing des arguments (commun Ex1/Ex2)

```
1 // Dans main(String[] args)
2 for (String a : args) {
3     if (a.startsWith("--src=")) {
4         td2.common.AstUtil.SOURCE_ROOT =
5             java.nio.file.Paths.get(a.substring("--src=".length()));
6     } else if ("--gui".equalsIgnoreCase(a)) {
7         showGui = true; // j'active la fen tre Swing
8     } else {
9         try { X = Integer.parseInt(a); } catch (NumberFormatException ignored)
10             {}
11     }
12 }
```

### Écriture du DOT (Ex2CallGraph)

```
1 String dot = toDot(graph);
2 System.out.println("\n=== DOT (GraphViz) ===");
3 System.out.println(dot);
4
5 java.nio.file.Path outDir = java.nio.file.Path.of("target", "reports");
6 java.nio.file.Files.createDirectories(outDir);
7 java.nio.file.Path dotFile = outDir.resolve("callgraph.dot");
8 java.nio.file.Files.writeString(dotFile, dot);
9 System.out.println("\n[DOT] crit : " + dotFile.toAbsolutePath());
```

### Comptage LOC (idée)

```
1 static long countLOCNoComments(String src) {
2     String noLine = LINE_COMMENT.matcher(src).replaceAll("");
3     String noBlock = BLOCK_COMMENT.matcher(noLine).replaceAll("");
4     return noBlock.lines().map(String::trim)
```

```

5         .filter(s -> !s.isEmpty()).count();
6     }

```

## 10 Liens du projet ( Github et Google Drive )

Lien GitHub :

<https://github.com/Rahma121-crtl/TP1-Partie2-EvolutionLogiciels-Final>

Lien de la vidéo de démonstration (Google Drive) :

<https://drive.google.com/file/d/1sgfY0oa3NMkgYLukfRZNwftJbrZNI6BJ/view?usp=sharing>

## 11 Projet métier complémentaire — company

Dans le prolongement des exercices sur les métriques et le graphe d'appel, j'ai conçu un **projet métier** intitulé **company**. Ce projet vise à illustrer la réutilisation de l'outil d'analyse statique développé dans un contexte applicatif plus concret.

### Structure du projet

Le package `company` contient trois classes principales :

- **Employee** : représente un employé avec un nom et un salaire, comportant les méthodes `work()` et `raiseSalary()`.
- **Employer** : représente un employeur capable d'embaucher (`hire()`), d'afficher la liste des salariés (`showAll()`) et de les faire travailler (`makeEveryoneWork()`).
- **CompanyTest** : contient la méthode `main()` qui orchestre les interactions entre `Employer` et `Employee`.

### Analyse du projet métier

L'outil `Ex2CallGraph` a été réutilisé sur ce package pour construire le **graphe d'appel statique** du projet métier. Le graphe obtenu met en évidence les appels entre les classes et les dépendances internes du code.

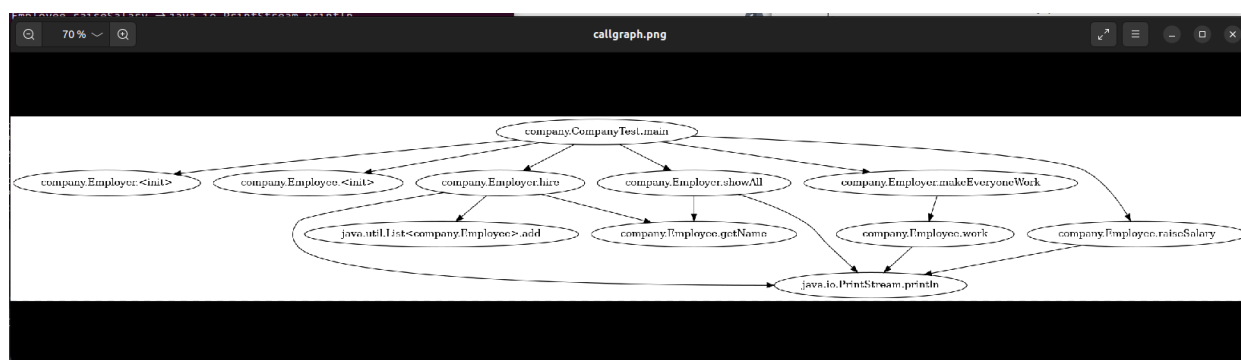


FIGURE 9 – Graphe d'appel du projet métier `company`.

### Interprétation du graphe

Le graphe d'appel montre :

- Les appels de `CompanyTest.main` vers les méthodes de `Employer` et `Employee`.
- Les interactions internes, telles que `Employer.showAll` → `Employee.getName` et `Employer.makeEveryoneWork` → `Employee.work`.

— Les dépendances vers la bibliothèque standard Java (`System.out.println`, `List.add`).

Ce projet métier démontre que l'outil développé est capable d'analyser automatiquement du **code Java réel**, et pas seulement les exemples académiques fournis dans le TP.