université de BORDEAUX

PdP - Métriques de Maintenabilité

# Mémoire

Jeudi 4 Avril 2019

Dépôt Savane :
`https://services.emi.u-bordeaux.fr/projet/savane/projects/pdp2019mm/`

*Soumis pour :*
(Client) Narbel Philippe
(Chargé de TD) Hofer Ludovic

*Soumis par :*
Delrée Sylvain
Giachino Nicolas
Martinez Eudes
Ousseny Irfaane

# Table des matières

# 1. Introduction

Lors du développement de systèmes logiciels de taille conséquente, se contenter d'en écrire le code ne suffit pas. En effet, sans exercer une réflexion sur la manière dont il est préférable de séparer et d'organiser les éléments de ceux-ci, il devient rapidement complexe d'en conserver une vue d'ensemble claire. Le processus de maintenance s'en trouve directement impacté. Il devient alors impératif d'élaborer une architecture pour de tels logiciels.

Dans ce contexte, étudier en profondeur les éléments qui semblent rendre un design flexible, robuste et maintenable ainsi que fournir des outils qui permettent l'extraction et l'analyse de ces éléments constitue une priorité. En effet, être capable de déterminer le degré de maintenabilité d'un projet permettrait aux développeurs d'effectuer un suivi de la qualité de leur application. Ceci rendrait le développement et la maintenance plus aisés.

Dans le cadre de l'unité d'enseignement « Projet de Programmation », la métrique définie par Martin[**?** ] va être au centre de notre projet. Cette dernière permet, au travers d'une analyse des dépendances, une étude sur la maintenabilité d'un projet de développement logiciel.

Dans le cas de notre projet, l'application reçoit en entrée un programme de paradigme Orienté Objet à analyser. L'application que nous devrons mettre en oeuvre devra réaliser une analyse au moyen de métriques logicielles afin d'obtenir des informations sur la maintenabilité en sortie.

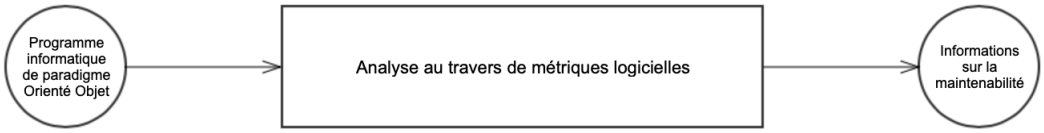


FIGURE 1 – Vision simplifiée du fonctionnement de l'application

# 2. Description du domaine

Cette section a pour vocation de décrire la métrique utilisée ainsi que tout le vocabulaire nécessaire à sa compréhension.

## 2.1. Préambule

Le domaine dans lequel s'inscrit l'application est celui de l'architecture logicielle. En effet, il s'agit ici de déterminer un indicateur du degré de qualité qu'un logiciel possède au regard de la manière dont sont structurés ses composants. Bien que de nombreuses métriques aient été élaborées dans ce but, le programme s'appuiera sur celle définie par Robert Martin[**?** ]. Cette étude s'intéressera à son application au paradigme orienté objet.

## 2.2. Concepts

### 2.2.1. Dépendances et granularités

**Granularité** La granularité *de structure* (par la suite nous utiliserons simplement le terme de granularité) est une échelle de groupement hiérarchique des éléments constitutifs d'un programme. Chaque niveau de granularité est une partition de l'ensemble de ces éléments. Des exemples de niveaux de granularité dans un programme orienté objet sont :

- Les attributs et méthodes.
- Les classes et objets.
- Les packages, les namespaces.
- Les super-packages.

Les éléments d'un niveau de granularité sont appelés **granules**. Chaque granule d'un niveau contient un ensemble de granules du niveau du dessous.

**Dépendance** Soient A et B des granules, le plus souvent du même niveau. A *dépend* de B dans le cas où A utilise B dans sa définition. Cette relation de dépendance peut s'exprimer sous différentes formes :

- **Dépendance par héritage** : A réutilise globalement B.
- **Dépendance par association** : A possède un attribut de type B.
- **Dépendance par utilisation** : A communique avec B.

**Couplage** Le degré de couplage est une mesure de l'*interdépendance* entre les différents granules d'un même niveau. On parle de couplage fort pour signifier que l'interdépendance est élevée.

**Cohésion** La cohésion représente le degré de liaison, de collaboration et d'interdépendance entre les éléments appartenant à un même granule. Une forte cohésion implique que le composant se concentre sur un seul et unique but, une seule et même responsabilité : réaliser des traitements relatifs uniquement à l'intention du composant.

**Principe de simplification min-max** Pour deux niveaux de granularité consécutifs, il faut favoriser une cohésion forte et un couplage faible.

### 2.2.2. Propriétés d'un design

Afin de comprendre la métrique détaillée dans ce document, il est nécessaire de définir plusieurs propriétés qu'une architecture logicielle peut posséder (ceux-ci sont repris de l'article fondateur de cette métrique, définie par Robert Martin[**?** ]) :

**Rigidité** Un design rigide est un design qui ne peut être facilement changé. C'est souvent le cas si les composants d'un système sont trop interdépendants. Dans ce cas, un changement dans un composant peut forcer beaucoup d'autres composants à changer également et son impact peut être difficile, si ce n'est impossible, à évaluer.

**Fragilité** Un design fragile est un design qui a tendance à cesser de fonctionner à plusieurs endroits si un seul changement est effectué. Dans la plupart des cas, les problèmes engendrés par cette modification surviennent à des endroits sans relation conceptuelle avec la partie ayant subie la modification. De plus, la correction de ces erreurs amène souvent à davantage de nouveaux problèmes.

**Robustesse** Un design robuste est l'exact opposé d'un design fragile. En effet, est considéré comme robuste un design au sein duquel un unique changement ne cause pas toute une cascade de problèmes.

**Maintenabilité** Un design maintenable est un design qui peut facilement évoluer. Il faut comprendre par là qu'il doit être facile d'ajouter de nouvelles fonctionnalités ou de modifier le comportement de celles déjà existantes. Un design rigide ou fragile sera peu maintenable.

**Réutilisabilité** Un design réutilisable est un design qui permet la réutilisation de certains de ses composants sans nécessiter d'embarquer ceux dont on ne veut pas. Si ses composants dépendent fortement les uns des autres, le design est dit difficile à réutiliser car il est compliqué d'isoler les composants désirés.

### 2.2.3. Propriétés d'un composant

Il s'agit ici de définir plusieurs propriétés que peuvent avoir les composants d'un logiciel. Ces propriétés interviennent dans le calcul de la métrique détaillée plus bas.

**Responsabilité** Un composant responsable est un composant dont dépendent d'autres composants. Un tel composant a intérêt à ne pas être souvent modifié car chaque changement peut se diffuser aux composants dépendant de lui.

**Indépendance** Un composant est dit indépendant s'il ne dépend d'aucun autre. Cette notion peut être nuancée : on peut admettre qu'un composant est très indépendant s'il dépend de peu d'autres et peu indépendant s'il dépend de beaucoup d'autres. Un composant très indépendant sera peu amené à changer à cause d'autres composants, de par son faible nombre de dépendances.

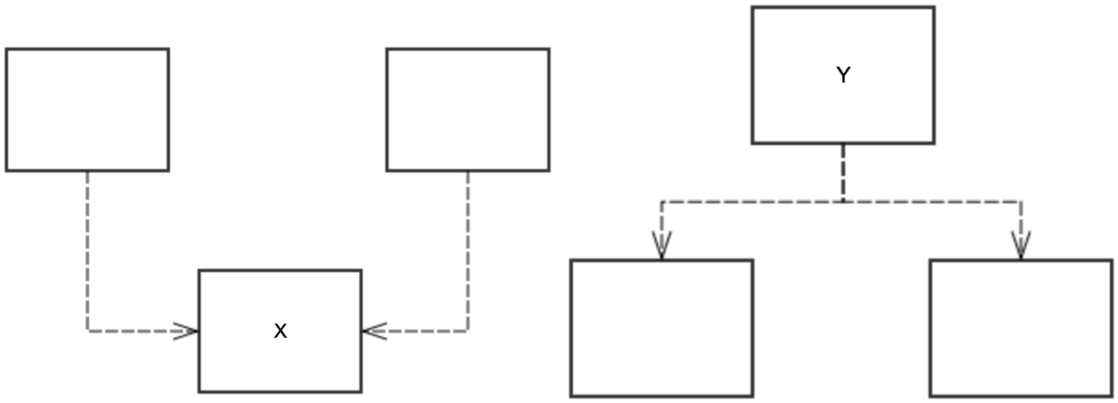


FIGURE 2 – Un granule stable X et un granule instable Y

**Stabilité** La stabilité est une propriété combinant les deux précédentes. En effet, joue en faveur de la stabilité d'un composant :

— Le nombre de composants dépendant de lui (*responsabilité*).
— L'absence de composants dont il dépend (*indépendance*).

La combinaison d'une forte responsabilité et d'une importante indépendance est synonyme de forte stabilité et vice versa. La stabilité vise à fournir une indication de la tendance qu'un composant aura à changer dans le temps. Plus le composant est stable, plus cette tendance est faible. En effet, un composant responsable aura peu tendance à changer à cause de la propagation des changements que cela peut engendrer. De même, un composant indépendant aura moins tendance à changer que s'il avait beaucoup de dépendances car une modification extérieure a peu de chances de se répercuter sur lui.

**Stable Dependency Principle (SDP)** Selon le SDP énoncé par Martin[**?** ], une bonne dépendance (*Good dependency*) est une dépendance dont la cible est très stable. Si A dépend de B et que B est très stable, alors cette dépendance est très bonne. Par opposition, une mauvaise dépendance (*Bad dependency*) est une dépendance dont la cible est instable. Ce sont, assez naturellement, ces dépendances qu'il faut éviter.

**Niveau d'abstraction** L'abstraction, ou le niveau d'abstraction, désigne la proportion d'un composant qui est abstraite (c.à.d. une partie ne comportant que des signatures et pas d'implémentation, dans le but d'être héritée et définie ailleurs). Plus un composant contient de parties abstraites par rapport à sa taille totale, plus il est lui-même abstrait.

**Stable Abstraction Principle (SAP)** Le SAP décrit par Martin[**?** ] est un principe qui met en relation les concepts de stabilité et d'abstraction. Ce principe stipule :

— qu'un composant stable doit également être abstrait pour que sa stabilité ne l'empêche pas d'être étendu (à comprendre ici dans le sens de fournir des alternatives, sous forme d'extensions horizontales proposant plusieurs implémentations distinctes).

— qu'un composant instable doit également être concret, car son instabilité permet de modifier facilement le code concret qu'il contient.

## 2.3. Métrique de Martin

### 2.3.1. Présentation et définitions

La métrique de Martin a été définie pour la première fois en 1994 par Robert Martin[**?** ]. L'auteur l'a par la suite citée au sein d'autres ouvrages[**?** ], et une large bibliographie scientifique mentionne, critique et complète celle-ci[**?** ][**?** ][**?** ][**?** ][**?** ]. La métrique s'articule autour de 2 notions centrales : la stabilité et le niveau d'abstraction (cf. 2.2.3).

**Composantes** Martin présente une métrique principale : la distance entre un *granule* représenté par un point de coordonnées (Instabilité, Abstraction) et la *Main Sequence* (définie plus précisément ci-dessous), une droite représentant le positionnement idéal des granules. Plus cette distance est grande, moins le granule correspond au modèle recherché. Afin de calculer cette distance, il est nécessaire de calculer plusieurs autres métriques, qui s'appliquent toutes à un granule :

— **Couplage Afférent** (*Afferent Coupling*), noté **Ca** : Il s'agit du nombre de granules externes qui dépendent du granule.

— **Couplage Efférent** (*Efferent Coupling*), noté **Ce** : Il s'agit du nombre de granules externes dont dépend le granule.

— **Instabilité** (*Instability*), notée **I** : Il s'agit d'une quantification de l'instabilité du granule (cf. 2.2) qui fait intervenir les métriques de couplage (Ca et Ce).

— **Niveau d'abstraction** (*Abstractness*), noté **A** : Il s'agit d'une quantification du niveau d'abstraction du granule.

— **Distance** (Distance par rapport à la Séquence Principale), notée **D** (ou **Dn** pour la version normalisée) : Il s'agit d'une mesure de la distance perpendiculaire du granule à la Séquence Principale. Cette distance donne une idée de la qualité du granule : le but est de minimiser cette valeur.

Il est également nécessaire de définir ce qu'est la **Séquence Principale** (*Main Sequence*). Elle intervient dans le contexte d'une représentation en deux dimensions des métriques. Dans ce plan, un granule est représenté par un point dont les coordonnées sont les suivantes : **(Instabilité, Niveau d'Abstraction)**. Les positions idéales se situent en coordonnées (0,1) : il s'agit d'un granule instable mais entièrement concret ; ainsi qu'en (1,0) : il s'agit d'un granule très stable et entièrement abstrait. Des compromis sont également possibles : il s'agit de la Séquence Principale, une droite passant par (0,1) et

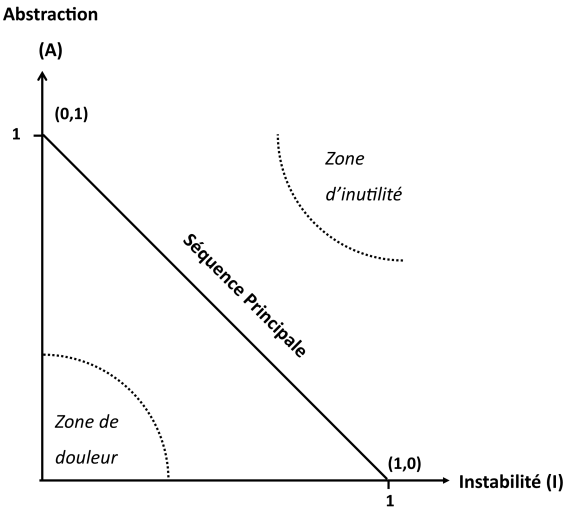


FIGURE 3 – Séquence principale

(1,0). Cette droite est la représentation du SAP. Les granules proches de cette droite sont donc considérées comme bien équilibrées selon ce principe.

### 2.3.2. Formalisation

**Catégorie de classe** La métrique exposé par Martin dans l'article *OO Design Quality Metrics : an Analysis of Dependency*[**?** ] publier en 1994, a pour but d'étudier les dépendances entres les différentes **catégories de classes** (*class categories*) d'un programme.

La notion de catégorie de classe est ici emprunté à Grady Booch[**?** ]. Une catégorie de classes est définit comme un ensemble de classes interdépendantes et cohésives partageant un but commun.

**Granularité package** En 2003, R. Martin redéfinit dans son livre[**?** ] la granularité sur laquelle s'applique la métrique. Le concept de catégorie de classe devient plus concret : il s'intéresse ici au niveau de granularité à l'échelle des packages. Il expose dans son livre différents principes d'organisation des packages tel que le SDP et le SAP qui sont des principes sous-jacent à la métrique. On considérera donc dans la suite de cette sous-section une granule comme étant un package.

**Composition de la métrique** La métrique de Martin est constitué des 5 composantes Ca, Ce, I, A et D définit précédemment. On appellera dans la suite de cette sous-section les composantes Ca, Ce, et A, les **composantes primordiales** de la métrique. Celles-ci doivent être calculés à partir des **données extractibles**. On appelle données extractibles, l'ensemble des données nécessaire à l'application de la métrique de Martin. Les composantes I et D sont déterminées à partir des composantes primordiales.

**Données extractibles** Les données extractibles sont représentées par le tuple $(G, A)$ définit sur $\mathbf{U}$ où :

— $G = (V, E)$, le graphe des dépendances où $V = \mathbf{U}$ et $E = \{i \to j \mid i \text{ dépend de } j\}$.

— $A \coloneqq (\mathbf{G}, \mathbf{NoC}(\mathbf{G}), \mathbf{NoAC}(\mathbf{G}))$ un 3-uplet représentant les données d'abstraction.

**Notations** On considère :

— $\mathbf{\Pi}$, un projet de programmation.

— $\mathbf{G} \in \mathbf{\Pi}$ un granule (package) appartenant au projet.

— $\mathbf{U} \coloneqq \{\mathbf{G} \mid \mathbf{G} \in \mathbf{\Pi}\}$, l'ensemble ou univers des granules composant le projet.

Afin d'exposer le calcul des composantes de la métrique on utilisera les notations :

— $\mathbf{DaC}(\mathbf{G}, \mathbf{U})$ = Le nombre de classes dans l'univers $\mathbf{U}$ de $\mathbf{G}$ dépendant de $\mathbf{G}$.

— $\mathbf{DeC}(\mathbf{G}, \mathbf{U})$ = Le nombre de classes dont dépend $\mathbf{G}$ dans son univers $\mathbf{U}$.

— $\mathbf{NoC}(\mathbf{G})$ = Le nombre de classes présentes au sein de $\mathbf{G}$.

— $\mathbf{NoAC}(\mathbf{G})$ = Le nombre de classes abstraites présentes au sein de $\mathbf{G}$.

**Calculs des composantes** Les différentes métriques définies plus haut sont ainsi calculables par application des formules suivantes :

— $\mathrm{Ca}(\mathbf{G}) = \mathbf{DaC}(\mathbf{G}, \mathbf{U}) \in \mathbb{N}$

— $\mathrm{Ce}(\mathbf{G}) = \mathbf{DeC}(\mathbf{G}, \mathbf{U}) \in \mathbb{N}$

— $\mathrm{A}(\mathbf{G}) = \mathbf{NoAC}(\mathbf{G}) \ / \ \mathbf{NoC}(\mathbf{G}) \in |0, 1|$

— $\mathrm{I}(\mathbf{G}) = \mathrm{Ce}(\mathbf{G}) \ / \ (\mathrm{Ca}(\mathbf{G}) + \mathrm{Ce}(\mathbf{G})) \in |0, 1|$

— $\mathrm{Dn}(\mathbf{G}) = |(\mathrm{A}(\mathbf{G}) + \mathrm{I}(\mathbf{G}) - 1)| \in |0, 1|$.

**Dépendances** La multiplicité de dépendance entre deux granules $\mathbf{G}_1$ et $\mathbf{G}_2$ est relative au nombre de dépendances distinctes entre $\mathbf{G}_1$ et $\mathbf{G}_2$. La métrique de Martin considère cette multiplicité dans les calculs effectués. En revanche, Martin ne fait pas allusion à l'importance (ou au type) des dépendances.

### 2.3.3. Critiques et pistes d'amélioration

Les résultats de la métrique de Martin — et des métriques logicielles en général — ne constituent pas une référence absolue. Leur intérêt est de fournir des indicateurs qui peuvent servir à détecter les zones d'un programme auxquelles prêter une attention particulière.

**Combinaison d'indicateurs** Dans l'optique de mesurer le degré de maintenabilité d'un logiciel, il est possible d'étudier le logiciel sous le prisme de différentes métriques. Par exemple, Chidamber et Kemerer ont défini en 1994 un ensemble de métriques[**?** ] qui permettent d'obtenir des indicateurs plus simples à étudier que ceux résultant de la métrique de Martin. Ces métriques portent sur l'étude du couplage et de la cohésion à différents niveaux de granularité. Dans cet ensemble, on trouve des métriques telles que : la profondeur de l'arbre d'héritage (métrique **DIT**), le nombre de filles d'une classe (métrique **NOC**) et le degré de cohésion des méthodes d'une classe (métrique **LCOM**).

> These metrics should assist software designers in their understanding of the complexity of their design and help direct them to simplifying their work. What the designers should strive for is strong cohesion and loose coupling.
>
> — C. Kemerer & S. Chidamber [**?** ] (p.229)

**Granularités** Comme explicité dans la section 2.3.2, Martin définit sa métrique comme s'appliquant à la granularité des packages. Cependant, les principes de stabilité et d'abstraction et leur mise en relation peuvent être étudiés à un niveau de granularité inférieur (classes et objets) ou supérieur (super-package).

**Mesure du niveau d'abstraction** La composante A (Abstractness) de la métrique de Martin est calculée à l'aide de la formule : **NoAC**(**G**) / **NoC**(**G**) (c.à.d. le rapport entre le nombre de classes abstraites et le nombre de classes totales présentes dans un package). Cette mesure peut-être raffinée afin d'être appliquée au niveau de granularité inférieur. En effet, soit **G** un granule représentant une classe. On considère :

- **NoM**(**G**) = Le nombre de méthodes présentes au sein de **G**.
- **NoAM**(**G**) = Le nombre de méthodes abstraites présentes au sein de **G**.

On peut alors définir le calcul de la composante A comme suit :

$$A(\mathbf{G}) = \mathbf{NoAM}(\mathbf{G}) \ / \ \mathbf{NoM}(\mathbf{G})$$

**Formalisme résultant** Si l'analyse porte sur plusieurs niveaux de granularité, les données extractibles peuvent être récupérées sur le niveau de granularité étudié le plus bas. Les données nécessaires au calcul des métriques sur les niveaux de granularité supérieurs pourront être déterminées à partir des données du niveau inférieur.

**Multiplicité** Comme explicité dans la partie Formalisation, Martin considère la multiplicité associée à une dépendance d'un granule vers un autre. Certains auteurs (tel que Spinellis[**?** ]) font l'impasse sur cette donnée dans leurs applications de la métrique de Martin, considérant ainsi qu'une dépendance d'un granule vers un autre est équivalente quelque soit son importance ou sa multiplicité.

**Pondération** Telle qu'elle est énoncée, la métrique de Martin considère les dépendances comme toutes égales. Or, une dépendance envers un granule stable ne devrait pas avoir le même poids qu'une dépendance envers un granule instable. Afin de raffiner le calcul des métriques, une analyse de la globalité du graphe des dépendances permettrait d'éliminer les dépendances stables du calcul d'instabilité (ou du moins de minimiser leur poids), en considérant qu'une dépendance stable ne rend pas le composant dépendant beaucoup plus instable. Il s'agit d'une exploitation du SDP qui, bien qu'énoncé par Martin, n'influe pas le calcul des métriques.

Il serait également possible d'appliquer une pondération relative à l'importance de la dépendance en fonction de son type. On a par ordre d'importance croissant : Dépendance par utilisation, Dépendance par association, Dépendance par héritage.

**SAP et Main Sequence** La définition de la Main Sequence (et donc du SAP) donnée par Martin est discutable sur plusieurs points :

— La position idéale en I = 0 et A = 1 est justifiée par le fait qu'un granule stable doive être abstrait pour pouvoir être étendu. Or, un granule concret peut également être étendu et il est même possible de réimplémenter la totalité de son comportement. L'extension d'un granule stable est donc toujours possible, quelle que soit son abstraction.

— La position indésirable en I = 1 et A = 1 est justifiée par le fait qu'il n'y a aucun intérêt d'avoir un granule abstrait sans dépendants. Cependant, certaines situations peuvent nécessiter un tel cas de figure. Par exemple, une API peut définir des interfaces qui ont vocation à être implémentées par les clients. Si l'API n'offre aucune implémentation de ces interfaces et les utilise assez peu, une analyse de celle-ci déterminerais que l'instabilité de telles interfaces est proche de 1, bien qu'il s'agisse d'un cas acceptable.

## 2.4. Application au langage Java

Notre application aura pour objectif d'effectuer les analyses nécessaire à l'application de la métrique de Martin sur des programmes écrits en langage Java. Pour faciliter ce travail, notre application sera elle-même développée en Java (version 8).

### 2.4.1. Granularité : Adaptation

Chaque langage de programmation propose ses propres structures de modularité. Ceci étant, il est nécessaire de préciser les structures auxquelles s'applique le concept de granularité en fonction du langage dans lequel est écrit le programme que l'on souhaite analyser.

**Granules** Les niveaux de granularité principaux qu'on retrouve en Java sont :

— les méthodes et attributs.

— les classes.

— les packages.

— les modules (Java, version supérieur à 8).

Un granule pourra donc désigner n'importe quel élément d'un de ces niveaux. Notre application ne s'intéressera cependant qu'aux niveaux "classes" et "packages".

### 2.4.2. Analyse de l'existant

Il existe différents projets qui abordent la question de la maintenabilité de logiciels au travers de l'étude de métriques (on notera que beaucoup d'entre eux exposent les métriques de Chidamber et Kemerer[**?** ]). Certains de ces outils s'intègrent à des IDE comme Eclipse ou IntelliJ, ce qui facilite grandement leur installation et leur utilisation.

**JHawk** JHawk[1] est un logiciel propriétaire d'analyse de code source Java. Il est en mesure de calculer un très grande nombre de métriques dont celle de Martin. Il peut être utilisé seul ou en tant que plugin Eclipse.

Il génère plusieurs métriques au niveau des méthodes, classes et packages. En ce qui concerne celle de Martin, il permet d'obtenir :

— Ca et Ce au niveau des classes.

— Ca, Ce, A, I et D au niveau des packages.

A cause du caractère propriétaire de JHawk, il est compliqué de déterminer comment il s'y prend pour ses calculs.

**JDepend** JDepend est un logiciel open source dont le code source est disponible sur la plateforme Github[2]. Il met en oeuvre la métrique de Martin (et nourrit donc des ambitions similaires à celles de notre application).

JDepend procède à une analyse statique des différents fichiers constituant un projet Java. Plus exactement, il analyse les fichiers compilés (`.class`) composés de bytecode (il peut également analyser les archives jar).

A la suite d'une étude du fonctionnement de JDepend, il ressort que celui-ci n'extrait que les informations de dépendance entre les packages. Il extrait les dépendances et calcule ensuite les valeurs de métriques à cette échelle. Il ne peut donc fournir aucune information sur les dépendances entre classes.

L'approche du projet décrit dans ce document est différente. En effet, là où JDepend se limite aux packages, notre application adopte une approche *bottom-up* : l'analyse s'effectue au niveau des classes (on ne considère pas la granularité des méthodes). A partir des résultats de celle-ci, on peut alors calculer les dépendances et métriques des échelles supérieures sans analyse supplémentaire.

Certains travaux se basent sur l'outil JDepend pour analyser des programmes, comme par exemple *Gephi*[3].

1. `http://www.virtualmachinery.com/jhawkprod.htm`
2. `https://github.com/clarkware/jdepend`
3. `https://dzone.com/articles/visualizing-and-analysing-java`

# 3. Expression des besoins

## 3.1. Besoins fonctionnels

**Évaluer les méthodes d'analyse et définir celle à utiliser**

- **Priorité** : Forte
- **Description** : En amont du développement de l'application, les méthodes d'extraction d'informations concernant les dépendances doivent être évaluées afin de déterminer si elles offrent les mêmes possibilités ainsi que la facilité à les mettre en place. Une comparaison est disponible en sous-section 4.3.1.

**Générer un ensemble de projets annotés (Création d'une vérité terrain)**

- **Priorité** : Forte
- **Description** : Afin de tester notre implémentation, il sera nécessaire de générer une série de projets Java et de fournir les métriques calculées manuellement. Dans ces différents cas d'exemples, seule la structure des dépendances (c.à.d. l'agencement des classes ainsi que les dépendances internes aux méthodes) est importante. Il sera donc inutile d'implémenter des fonctionnalités spécifiques.

### 3.1.1. Sélection et structuration de l'entrée

**Sélectionner un projet depuis un répertoire local**

- **Priorité** : Forte
- **Description** : L'utilisateur doit pouvoir renseigner un projet en entrée de l'application depuis le système de fichiers de la machine, sous forme de chemin d'accès au répertoire racine de celui-ci.

**Lister récursivement le contenu d'un répertoire**

- **Priorité** : Forte
- **Description** : Pour un répertoire donné, l'application doit être en mesure de lister son contenu. Dans le cas d'une analyse du code source (respectivement, d'une analyse du bytecode), l'application devra pouvoir lister l'arborescence des différents fichiers `.java` (respectivement `.class`). Si le répertoire donné ne contient pas un projet Java (sous la forme d'un ensemble de `.class` ou de `.java`), l'application devra renvoyer une erreur.

**Créer une structure représentative de l'organisation du projet**

- **Priorité** : Forte
- **Description** : L'application devra créer une structure arborescente contenant tous les packages du projet analysé ainsi que les classes qui les composent.

### 3.1.2. Analyse de fichiers

L'application doit être en mesure de réaliser une **analyse statique** de fichiers. L'analyse de fichiers consistera en la mesure du niveau d'abstraction des classes ainsi que l'extraction des dépendances. Lors de l'analyse, nous ne nous intéresserons qu'à l'ensemble des dépendances sortantes ; nous pourrons déterminer les dépendances entrantes à partir des premières. Cette mesure permettra de déterminer les composantes Ce et A.

**Extraire les dépendances par héritage/implémentation**

- **Priorité** : Forte
- **Description** : L'application devra extraire les dépendances de type héritage/implémentation. Cette information se trouve dans la déclaration de la classe.

**Extraire les dépendances par association**

- **Priorité** : Forte
- **Description** : L'application devra extraire les dépendances de type agrégation/composition. Cette information se trouve dans la liste des attributs de la classe. Cela implique que l'application devra être en mesure de lister les attributs d'une classe et leurs types.

**Extraire les dépendances d'utilisation**

- **Priorité** : Moyenne
- **Description** : L'application devra extraire les dépendances de lien d'utilisation. Il s'agit des variables locales, des appels de méthode statique, des paramètres ou des valeurs de retour d'une méthode.

**Extraire les dépendances liées à la généricité**

— **Priorité** : Faible
— **Description** : L'application devra extraire les dépendances dues à la généricité. Cette information peut se trouver dans la déclaration de la classe, dans la signature ou dans le corps des méthodes. Cette dépendance est plus compliquée à analyser car elle peut prendre plusieurs formes.

**Mesurer le nombre de méthodes d'une classe**

— **Priorité** : Forte
— **Description** : Afin d'implémenter la fonction **NoM**, il faudra mettre en place une méthode permettant de compter le nombre de méthodes qu'une classe définit.

**Mesurer le nombre de méthodes abstraites d'une classe**

— **Priorité** : Forte
— **Description** : Afin d'implémenter la fonction **NoAM**, il faudra mettre en place une méthode permettant de compter le nombre de méthodes abstraites qu'une classe définit.

### 3.1.3. Exploitation de la métrique

**Calculer le couplage afférent (Ca)**

— **Priorité** : Forte
— **Description** : A partir du couplage efférent (Ce) de toutes les classes, l'application devra déterminer le couplage afférent (Ca) de chaque classe en examinant le nombre de classes ayant des dépendances sortantes vers cette classe.

**Calculer la composante d'instabilité de la métrique**

— **Priorité** : Forte
— **Description** : L'application devra être en mesure d'effectuer le calcul de la composante I de la métrique pour une classe donnée à partir de la formule définie par Martin : I(**G**) = Ce(**G**) / (Ca(**G**) + Ce(**G**).

**Calculer la composante de distance de la métrique**

— **Priorité** : Forte
— **Description** : L'application devra être en mesure d'effectuer le calcul de la composante Dn de la métrique pour une classe donnée à partir de la formule définie par Martin : Dn($\mathbf{G}$) = |(A($\mathbf{G}$) + I($\mathbf{G}$) - 1)|.

**Générer un graphe de dépendances**

— **Priorité** : Forte
— **Description** : A partir des dépendances extraites de l'analyse de fichiers, l'application devra pouvoir générer une structure de données représentant l'inter-dépendance entre les différentes classes du projet : un graphe de dépendances. Le graphe de dépendances est défini comme un graphe orienté composé d'un ensemble de $n$ noeuds représentant les $n$ classes de l'analyse et d'un ensemble de $p$ arcs représentant les dépendances entre celles-ci. Étant donné que l'application fait la différence entre chaque type de dépendance, il peut y avoir plusieurs arcs d'un noeud vers un autre (un par type de dépendance).

**Calcul de métrique par granularité**

— **Priorité** : Forte
— **Description** : A partir de ce graphe de dépendances de classes, il est possible de passer à un niveau d'échelle supérieur en fusionnant les noeuds pour les regrouper par granule de niveau supérieur (package) et en ne conservant que les arcs sortants et entrants dans ces super-noeuds.

**Générer des tableaux exposant les composantes de la métrique**

— **Priorité** : Forte
— **Description** : L'application devra être en mesure de créer des tableaux exposant les différentes composantes de la métrique, celles récupérées au travers de l'analyse de fichiers (Ce et A) et celles calculées par la suite (Ca, I et D). On pourra obtenir les informations relatives au changement d'échelle à partir des données du graphe de dépendances.

### 3.1.4. Réalisation de rapport d'analyse

L'application devra permettre de générer des fichiers exposant les différentes métriques qu'elle aura traité. Ces fichiers pourront ensuite être interprétés par des outils externes.

**Générer des fichiers DOT exposant le graphe des dépendances**

- **Priorité** : Forte
- **Description** : L'application devra être en mesure de générer des fichiers au format DOT. Ces fichiers contiendront les graphes de dépendance calculés précédemment. Il y aura un fichier par échelle d'analyse (catégories). Les noeuds contiendront le nom des catégories (nom de classe, de package,...) associé à leur valeur d'instabilité. Ces fichiers pourront ensuite être interprétés par des outils tels que GraphViz.

**Générer des fichiers CSV exposant les métriques**

- **Priorité** : Forte
- **Description** : L'application devra être en mesure de générer des fichiers au format CSV. Ils contiendront les valeurs des différentes composantes de la métrique. Tout comme les fichiers DOT, il en existera un par échelle d'analyse. Ces fichiers pourront ensuite être interprétés par des outils tels que Libre Office Calc.

**Générer une sortie matricielle du graphe**

- **Priorité** : Faible
- **Description** : L'application devra être en mesure de générer des fichiers au format CSV exposant une représentation du graphe sous forme de matrice d'adjacence.

### 3.2. Besoins non fonctionnels

**Documentation**

— **Priorité** : Forte
— **Description** : La documentation de l'application sera scindée en deux documents : l'architecture générale et le manuel d'utilisation. Ceux-ci seront produits dans le format LaTeX afin de permettre leur intégration dans le rapport terminal de l'UE.
**Architecture générale** : Pour aider les développeurs à modifier / adapter / ajouter des fonctionnalités à l'application.
**Manuel d'utilisation** : Pour aider l'utilisateur à prendre en main l'application.

**Modularité**

— **Priorité** : Forte
— **Description** : Les algorithmes de calcul des métriques doivent pouvoir être aisément modifiés. L'application doit adopter une architecture lui permettant de s'adapter sans nécessiter de changements conséquents.

**Configuration**

— **Priorité** : Faible
— **Description** : La possibilité de modifier certains paramètres de l'application (par exemple, il pourrait être possible de paramétrer une pondération à appliquer à chaque type de dépendance dans le calcul de la métrique) à l'aide d'un fichier de configuration peut être implémentée en guise d'alternative à la modification directe d'un composant dans le code. La présence d'une telle fonctionnalité est optionnelle.

# 4. Architecture

Dans une première sous-partie, nous allons vous présenter le pipeline de notre projet, qui correspond à la chaîne de traitement de notre application à l'échelle des packages. Nous rentrerons ensuite plus en détails dans chaque package en présentant les différentes classes décrivant notre architecture. Finalement, nous traiterons les spécificités techniques qui ont motivé nos choix d'implémentation.

## 4.1. Pipeline de traitement

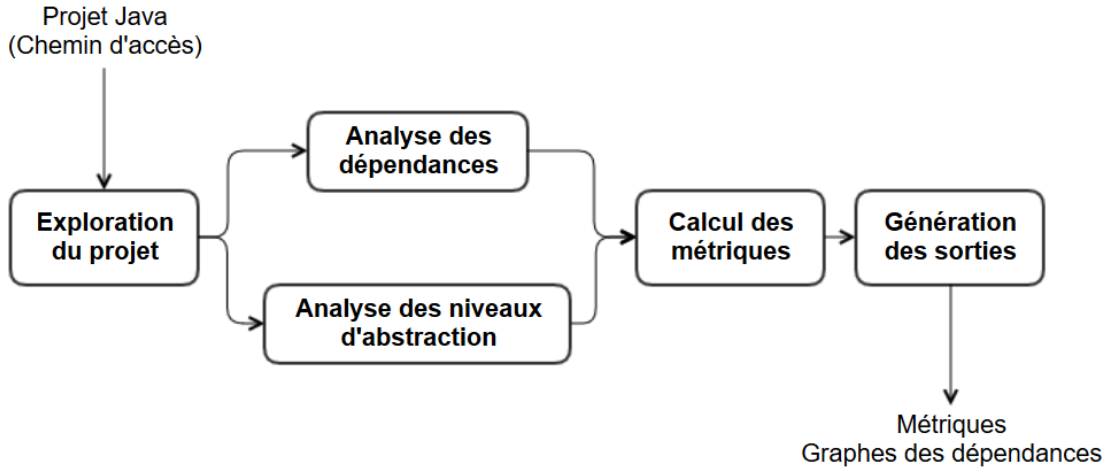


FIGURE 4 – Pipeline de traitement de JMetrics

Afin que l'application atteigne son objectif d'obtention de métriques, il est nécessaire de passer par plusieurs étapes afin d'obtenir les informations nécessaires à leur calcul. On peut représenter ces différentes actions sous la forme d'un pipeline de traitement. En effet :

— Chaque étape nécessite le résultat de la précédente.
— Aucun retour en arrière n'a lieu : une fois une action effectuée, il n'est plus nécessaire d'y revenir plus loin dans l'exécution.

La figure 4 illustre le pipeline dans les grandes lignes (il s'agit d'une version plus détaillée de la figure 1 présente dans l'introduction). L'application est architecturée de manière à ce que chaque package ait la responsabilité de l'exécution d'une étape, chacune ne nécessitant qu'une interaction très limitée avec les autres. Ce pipeline se décompose en quatre phases principales :

**Exploration du projet (Package `project`)** Dans un premier temps, l'application a besoin de construire une représentation exploitable du projet à analyser. Pour ce faire, celle-ci visite récursivement les répertoires et fichiers composant ce projet et crée une structure arborescente le représentant. La structure ainsi obtenue est ensuite nettoyée afin d'écarter les dossiers vides ou n'ayant que peu d'intérêt (par exemple un dossier n'en contenant qu'un autre).

**Analyse statique des classes (Package `analysis`)** A partir des structures de données précédemment générées, le programme parcourt toutes les classes référencées dans le projet et effectue une analyse de leur code (source ou bien compilé, en fonction du type d'analyse choisi) afin d'en extraire les données nécessaires au calcul des métriques. Cette étape peut se scinder en deux sous-parties indépendantes et ayant la possibilité d'être effectuées en parallèle :

— *Analyse du niveau d'abstraction* : Il s'agit d'effectuer un comptage des méthodes composant la classe et d'en tirer deux valeurs : le *nombre total de méthodes* et le *nombre de méthodes abstraites* de la classe.

— *Analyse des dépendances* : Cette seconde partie est la plus délicate des deux : il s'agit d'extraire les dépendances de la classe vis-à-vis des autres classes du projet. Cette analyse génère une *liste de ces dépendances*.

A partir du résultat de l'analyse des dépendances, l'application génère un graphe pour représenter celles-ci.

**Calcul des métriques (Package `metrics`)** En utilisant les données renvoyées par l'étape d'analyse, l'application calcule différentes métriques (pour le moment : les métriques de Martin, mais avec possibilité d'extension) pour chaque classe, puis les classes sont regroupées en packages et les valeurs des métriques pour chacun d'eux sont calculées à partir des résultats précédents. Bien que ce ne soit pas le cas pour le moment, une approche similaire pourrait être envisagée pour d'autres niveaux de granularité.

**Présentation des résultats (Package `presentation`)** Pour finir, les valeurs calculées précédemment doivent être présentées à l'utilisateur. Dans ce but, l'application génère plusieurs fichiers contenant les informations nécessaires (par exemple : graphe de dépendances au format DOT, tableau de métriques au format CSV, ...).

Afin de faciliter la visualisation des données générées par l'application, nous avons mis en place plusieurs scripts *Python*. Ils ne sont cependant que des aides, ne font pas partie du cœur de l'application et ne sont donc pas considérés comme une étape à part entière du pipeline. En outre, un cinquième package (`graph`) fait partie du projet mais n'a vocation qu'à fournir une structure de graphe partagée entre les autres packages. Il ne constitue donc pas une étape de traitement non plus.

## 4.2. Organisation des composants

### 4.2.1. Package project

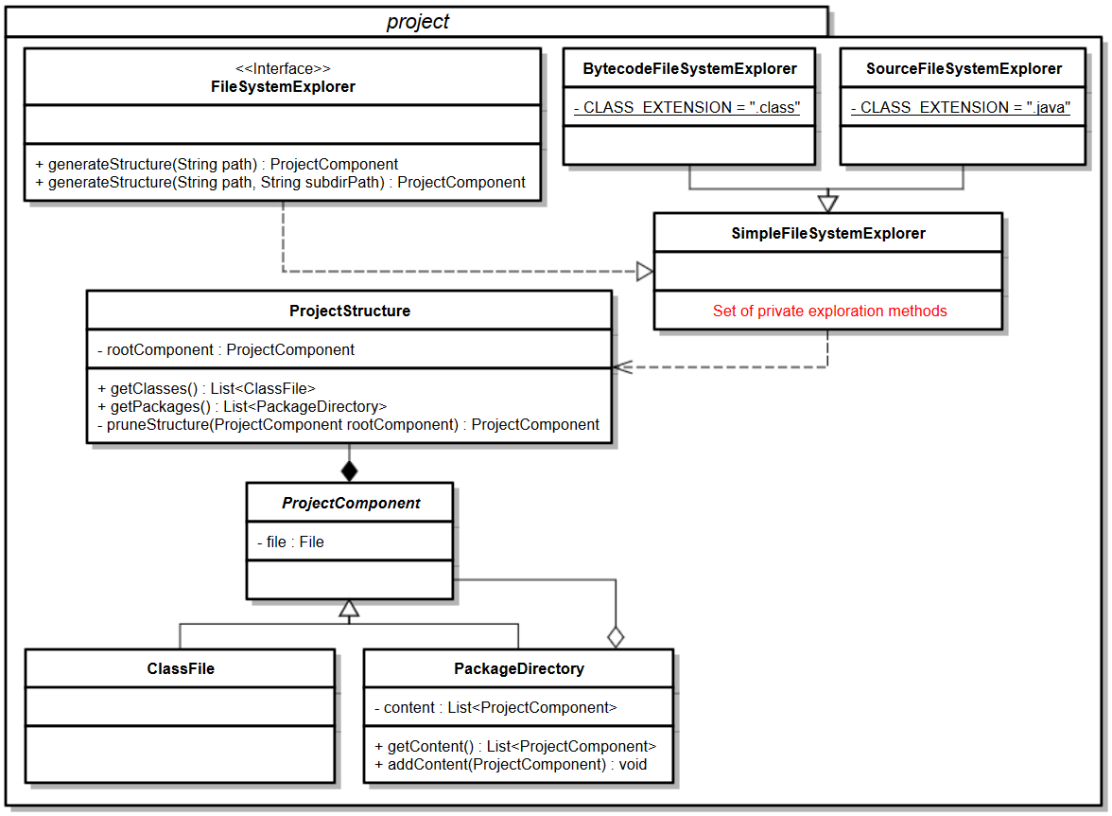


FIGURE 5 – Diagramme de classe (UML) du package `project`

Le package `project` fournit les classes nécessaire à l'exploration d'un répertoire donnée et à la création d'une représentation de projet Java.

Le package est composé des trois éléments suivants :

— Une instance du pattern Composite permet de représenter l'arborescence des composants du projet. Une classe mère `ProjectComponent` maintient une référence en tant qu'attribut vers un objet `File`. Les packages sont représentés par la classe composite `PackageDirectory` exposant un ensemble de `ProjectComponent`. Les classes sont représentées par la classe feuille `ClassFile`.

— La `ProjectStructure` est une classe Singleton qui conserve en mémoire la structure du projet qui s'apprête à être analysé et qui fournit des méthodes permettant l'accès

aux différents composants de ceuil-ci (classes et packages). Elle représente donc un repository.

— Enfin, le `FileSystemExplorer` est un service qui a pour objectif de construire la représentation du projet en explorant le système de fichiers et d'affecter la racine de celle-ci dans la classe `ProjectStructure`. Notre implémentation du FileSystemExplorer est composée de quatre classes :
  — Une interface `FileSystemExplorer` qui permet de définir les méthodes d'exploration.
  — Une classe abstraite `SimpleFileSystemExplorer` qui constitue le coeur du processus d'exploration. Elle contient toutes les méthodes nécessaire à cette tâche, permettant une factorisation importante du code. Le code qu'elle contient est capable de lister tous les fichiers possédant une certaine extension (à définir dans les classes filles).
  — Enfin, deux classes concrètes `BytecodeFileSystemExplorer` et `SourceFileSystemExplorer` qui définissent toutes deux les extensions des fichiers à explorer (*.class* et *.java*).

La structure du projet est construite et parcourue à l'aide d'algorithmes récursifs.

### 4.2.2. Package analysis

Le package `analysis` contient les composants chargés d'effectuer l'analyse statique du code des classes. Cette analyse est menée à bien par des parseurs. Il en existe deux catégories, chacune définie par une interface :

— **AbstractnessParser**, parseur d'abstraction. Définit l'interface que doivent implémenter tous les parseurs ayant vocation à extraire les informations sur le niveau d'abstraction d'une classe (nombre de méthodes et nombre de méthodes abstraites).
— **CouplingParser**, parseur de dépendances. Définit l'interface que doivent implémenter tous les parseurs qui récupèrent les dépendances d'une classe envers les autres.

Le parseur d'abstraction retourne un objet de la classe `AbstractnessData`. Cette classe ne possède que deux attributs (`numberOfMethods` et `numberOfAbstractMethods`) et les *getters* permettant d'y accéder. Le but de cette classe est uniquement d'encapsuler ces éléments. Le parseur de dépendances, quant à lui, retourne une liste d'objets de la classe `Dependency`. Cette classe a également pour but d'encapsuler les informations relatives à une dépendance (`source`, `destination` et `type`). Il s'agit d'une représentation intermédiaire avant la construction du graphe de dépendances. Dans son état actuel, l'application contient deux implémentations de chacune de ces interfaces : une permettant l'analyse du code source via *l'API Eclipse JDT*[4] et l'autre servant à l'analyse du code compilé via *l'API d'introspection* `java.lang.reflect`[5]. Chaque implémentation est constituée de trois classes :

4. `https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Foverview-summary.html`
5. `https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html`

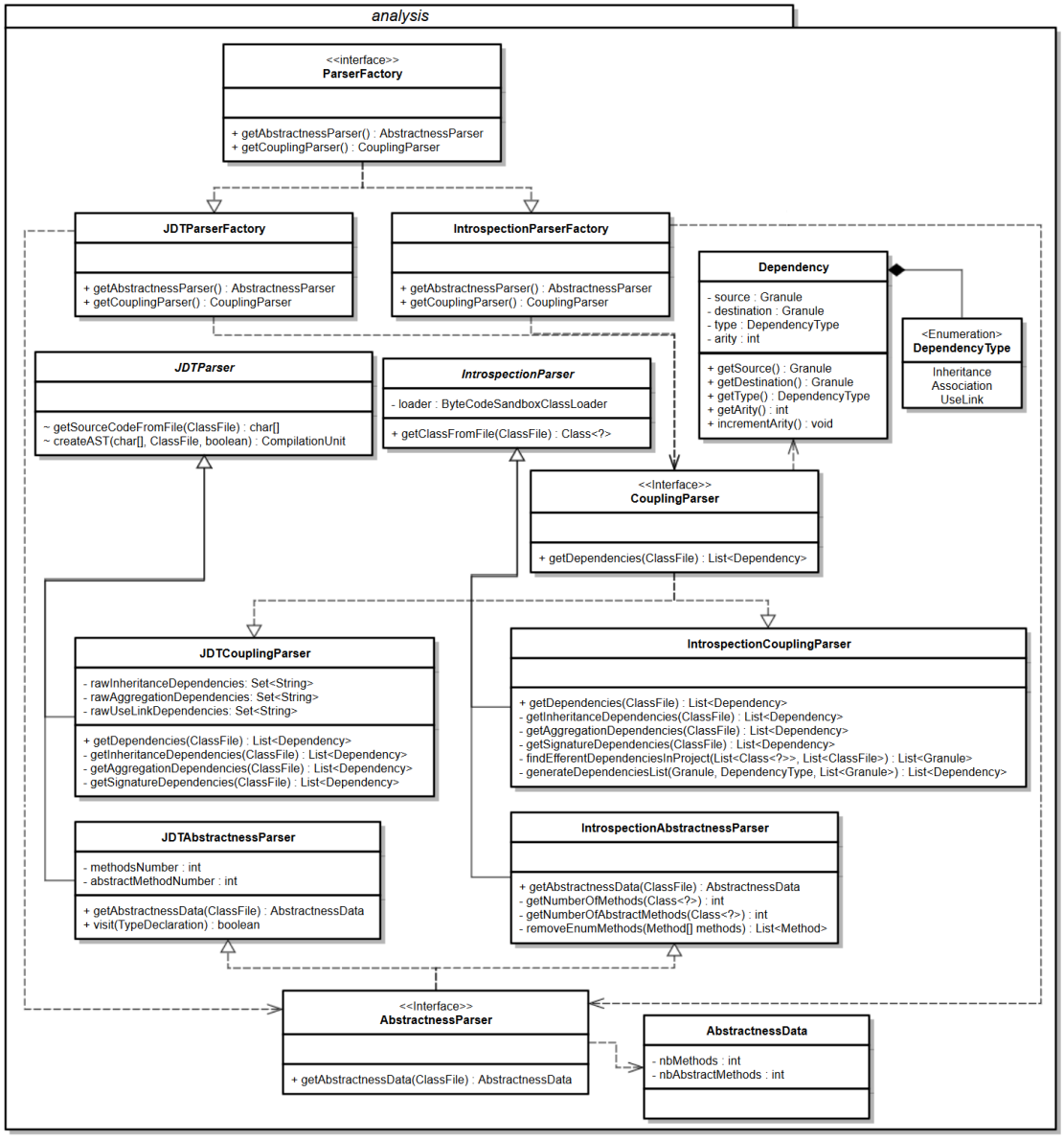


FIGURE 6 – Diagramme de classe (UML) du package `analysis`

— Une implémentation d'`AbstractnessParser`. Il s'agit de `IntrospectionAbstractnessParser` pour l'introspection et `JDTAbstractnessParser` pour l'implémentation JDT.
— Une implémentation de `CouplingParser`. Il s'agit de `IntrospectionCouplingParser` pour l'introspection et `JDTCouplingParser` pour l'implémentation JDT.
— Une classe abstraite pour factoriser le code partagé. Il s'agit de `Introspection-`

`Parser` pour l'introspection et `JDTParser` pour l'implémentation JDT.
L'implémentation utilisant le JDT a une particularité supplémentaire. En effet, l'API JDT faisant usage du pattern *Visitor* pour parcourir le code source, les deux parseurs implémentent ce pattern. Il existe donc une méthode *visit* pour chaque partie du code source permettant de récupérer les informations recherchées.

Afin de faciliter l'instanciation des parseurs par le code client, une interface `ParserFactory` est définie et implémentée pour fournir une *factory* pour chacune des deux familles de parseurs (introspection et JDT) : `IntrospectionParserFactory` et `JDTParserFactory`. Pour le code client, il est obligatoire de passer par une de ces factory car les constructeurs des parseurs ne sont disponibles que dans leur propre package.

Enfin, ce package définit également les différents types de dépendances (*Inheritance*, *Association* et *UseLink*) au sein de l'énumération `DependencyType`.

### 4.2.3. Package graph

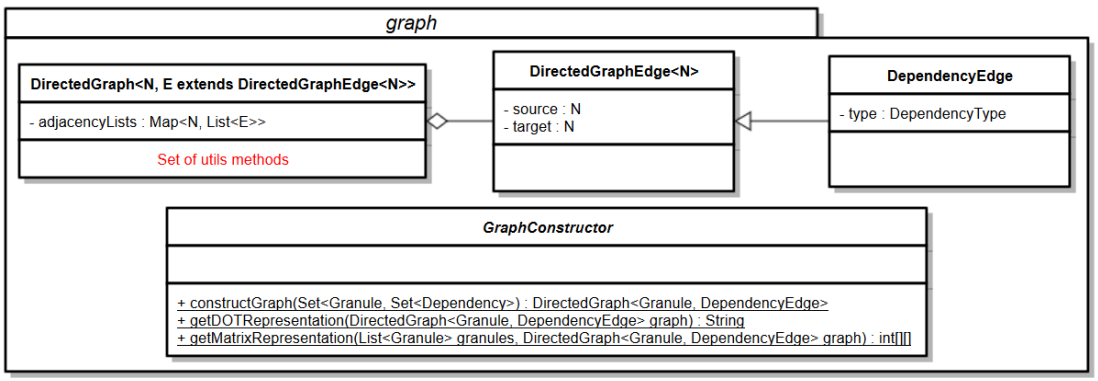


FIGURE 7 – Diagramme de classe (UML) du package `graph`

Le package `graph` a pour objectif de mettre à disposition de l'application un graphe structurant un ensemble de dépendances et offrant différents algorithmes permettant d'effectuer des calculs sur ce graphe. Ce package est composé d'une classe principale `DirectedGraph` qui définit un graphe orienté, implémenté au moyen de listes d'adjacence. Les noeuds et arrêtes du graphe sont des types génériques.
Le service `GraphConstructor` est quand à lui chargé de construire un graphe à partir d'un ensemble de `Granule` et d'un ensemble de dépendances entre ceux-ci. Il assure également la construction de la représentation au format *.dot* d'un graphe orienté et sa conversion en forme matricielle (matrice d'adjacence).
La classe `DirectedGraphEdge` définit la structure de données utilisée pour stocker les informations concernant les arêtes. La classe `DependencyEdge` est une extension de

celle-ci contenant des informations supplémentaires relatives aux dépendances (type et multiplicité).

### 4.2.4. Package metrics

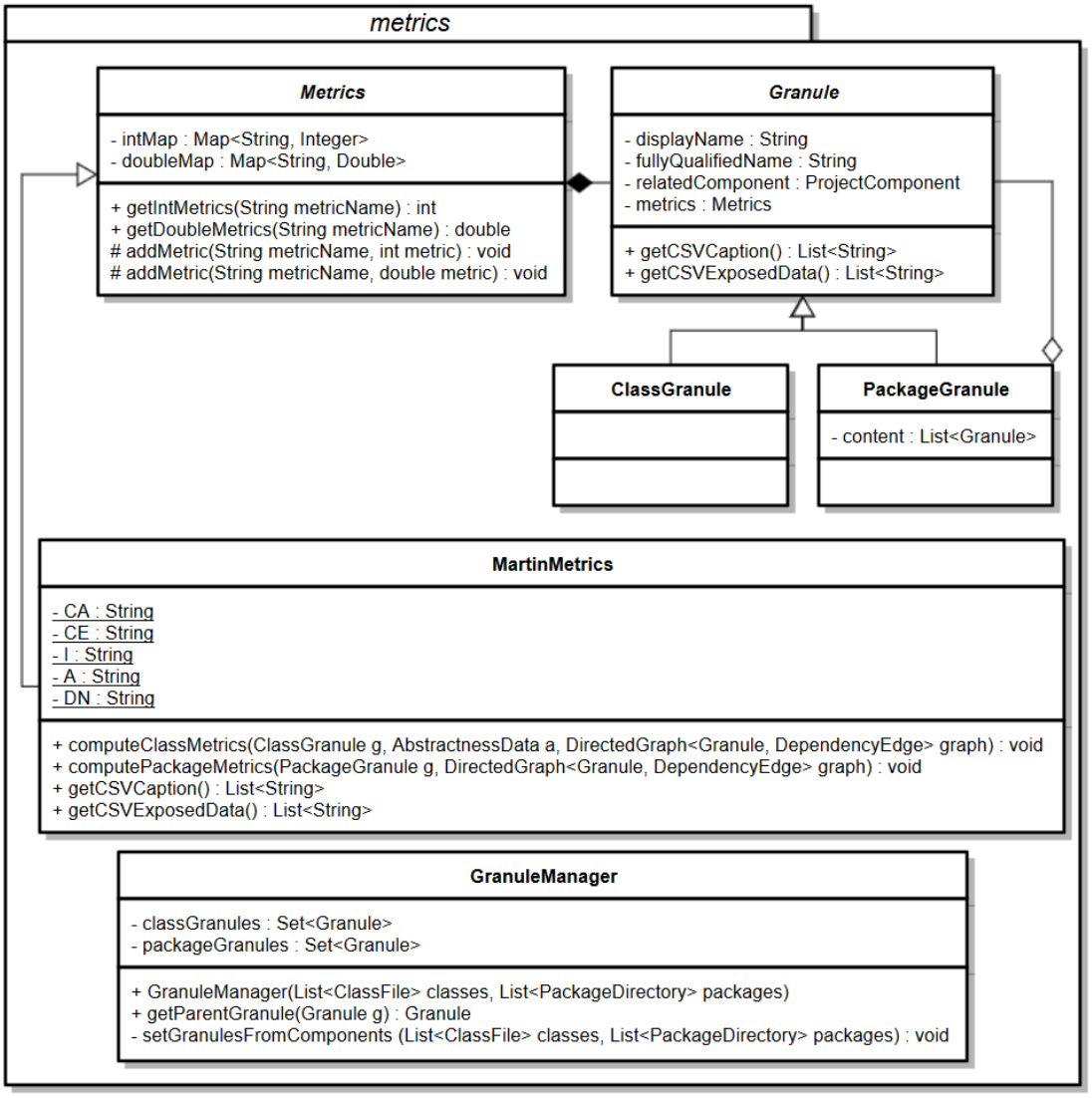


FIGURE 8 – Diagramme de classe (UML) du package `metrics`

Le package `metrics` contient les classes permettant de réaliser le calcul de métriques selon le niveau de granularité (tels que les packages et les classes), ainsi que des classes permettant de représenter les granules.

Les classes `Granule`, `ClassGranule` et `PackageGranule` forment une instance du pattern composite générée à partir des données fournies par le package `project`. Les formes des deux structures sont très similaires, l'intérêt de passer de l'une à l'autre est de se débarrasser des références vers les fichiers de code. La conversion entre celles-ci est effectuée par l'intermédiaire du service `GranuleManager`. La structure du package `metrics` permet de hiérarchiser les différents granules du projet (par niveaux de granularité).

La classe abstraite `Granule` représente ainsi les composants à analyser. Implémentant l'interface `CSVRepresentable`, elle fournit les méthodes relatives à l'exportation des résultats des métriques au format CSV. La représentation des métriques est déléguée à leur implémentation. Les classes `ClassGranule` et `PackageDirectory` représentent respectivement une classe et un package et contiennent leurs métriques. La classe `GranuleManager` fournit des services permettant de gérer les différents granules d'un projet.

L'ensemble des métriques décrites par Martin[**?** ] est implémenté par la classe `MartinMetrics` qui hérite de la classe abstraite `Metrics`. Cette classe abstraite permet d'apporter des possibilités d'extension. Ainsi, il sera possible pour un futur utilisateur d'ajouter ses propres métriques. Le stockage des différentes composantes des métriques est permis grâce à deux `HashMap`, dont le tuple (clé, valeur) représente le nom et la valeur de la composante. Leur différence se trouve dans le type de la valeur en question ; l'une d'entre elles prendra des `int` comme valeur alors que l'autre prendra des `double`.

### 4.2.5. Package presentation

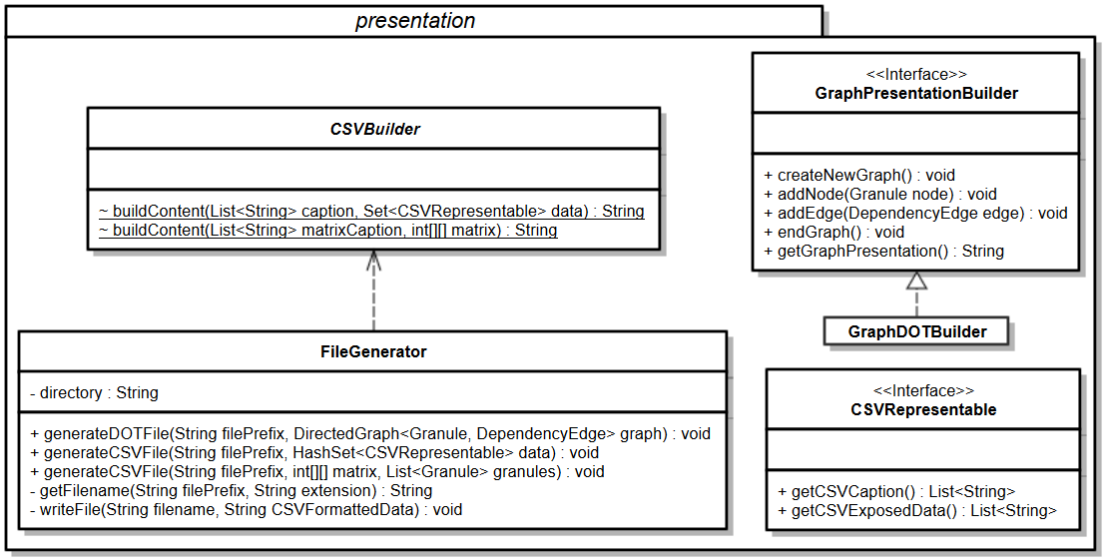


FIGURE 9 – Diagramme de classe (UML) du package `presentation`

Le package `presentation` fournit les classes nécessaires pour générer différents rapports d'analyse sous forme de fichiers.

A cette fin, nous avons mis en place deux interfaces permettant la représentation des structures de données de l'application :

— `GraphPresentationBuilder` : Définit les méthodes de représentation de graphe des dépendances sous la forme d'une chaîne de caractères.
— `CSVRepresentable` : Définit les méthodes permettant de représenter un objet au format CSV. L'interface définit les méthodes `getCaption` qui permet de fournir la légende du fichier CSV et la méthode `getExposedData` qui permet de fournir les données de l'instance qui seront exportées. Ces deux méthodes retournent une `List` de `String`.

Dans l'ensemble de notre projet, nous trouvons trois classes qui implémentent l'interface `CSVRepresentable` :

— `Granule` : Expose le nom du granule et l'ensemble des valeurs des métriques qui lui sont associées.
— `Metrics` : Expose le nom des composantes de la métrique ainsi que ses valeurs. Les méthodes de l'interface `CSVRepresentable` ne sont pas directement définies dans la classe abstraite : ce sont les classes qui héritent de celles-ci qui auront pour rôle de les implémenter.
— `Dependency` : Expose le granule source, le granule destination, le type et l'arité de la dépendance.

A partir d'un ensemble d'instances de `CSVRepresentable`, le service `CSVBuilder` va construire une chaîne de caractères au format CSV.

Une implémentation de l'interface `GraphPresentationBuilder` est donnée dans notre projet par la classe `GraphDotBuilder` qui aura pour but de construire la représentation du graphe des dépendances dans le format DOT.

Enfin, la génération des fichiers (au format DOT et CSV) sera réalisée par l'intermédiaire de la classe `FileGenerator`.

## 4.3. Choix d'implémentation

### 4.3.1. Comparaison des méthodes d'analyse

Il existe deux méthodes pour analyser un programme Java : l'analyse de bytecode et l'analyse de code source. Le but de cette section est de dresser une comparaison entre celles-ci.

**Informations récupérables** Tout d'abord, le code source contient des informations perdues à la compilation et qui ne peuvent donc pas être retrouvées dans le bytecode. En effet, toutes les informations liées aux types génériques sont effacées car elles ne sont utilisées que dans le but d'effectuer des vérifications à la compilation, empêchant la détection de certaines dépendances.

Par exemple, si on dispose du programme suivant :

```java
public class A {

}
```

```java
public class B {
    private List<A> aList;
}
```

On remarque que B possède une liste de A (il s'agit donc d'une dépendance d'association, B ayant une agrégation de A). Lorsque l'analyse passera sur l'attribut `aList`, on s'attend à ce que son type soit `List<A>` afin de pouvoir extraire la dépendance vers A. Si on effectue l'analyse sur le code source, on obtient effectivement ce résultat. Si cependant on analyse le bytecode du même programme, le type de `aList` sera uniquement `List`, l'information de généricité du type `List` étant effacée à la compilation. En utilisant la méthode analysant le bytecode, aucune dépendance de B vers A ne sera donc détectée sur ce programme. L'analyse de code source n'a pas cette limitation car les informations de types génériques y sont écrites.

Certaines autres indications sont également absentes du bytecode : la déclaration de nom de package (bien que récupérable depuis le nom complet de la classe), les déclarations *import*, ... Ceci n'est pas gênant car l'application n'exploite pas ces informations.

**Outils et facilité de mise en place** Il existe plusieurs outils/bibliothèques permettant d'exploiter l'une des deux méthodes :

**Bytecode** Les principales bibliothèques sont les suivantes :

*API java.reflect* L'introspection est la manière la plus rapide d'extraire des données depuis un code compilé car il s'agit d'une API directement intégrée au JDK. Le chargement des classes est facile et le parcours des attributs et méthodes également. Il est cependant plus difficile de charger une classe et l'analyser via cet API si cette même classe est déjà chargée dans la JVM et que les versions diffèrent (par exemple, si on analyse l'API Java 5 en lançant le locigiel sur un JVM Java 8 : beaucoup de classes déjà chargée en mémoire portent le même nom mais comportent énormément de différences). Ceci est dû au fait que pour récupérer des informations sur un classe via l'introspection, il est nécessaire de la charger dans la JVM comme s'il s'agissait d'une classe faisant partie de notre logiciel. En outre, cette API ne permet pas l'accès au corps des méthodes, empêchant par là même l'extraction de la grande majorité des dépendances de liens d'utilisations.

6. `https://asm.ow2.io/`

*ASM*[6] La bibliothèque permettant l'analyse (et même la modification) de bytecode la plus populaire est ASM. Cette dernière, contrairement à l'introspection, ne nécessite pas de charger les classes dans le JVM et permet l'accès aux instructions situées dans le corps des méthodes. Il s'agit cependant d'une bibliothèque externe et son utilisation nécessite d'implémenter certaines de ses interfaces (en suivant notamment le pattern Visitor) afin d'accéder au code, ce qui rend son adoption un peu moins rapide que celle de l'introspection. Cependant, elle permet d'extraire toutes les informations contenues dans le bytecode, permettant ainsi de récupérer la plupart des dépendances.

**Code source** Les principales bibliothèques sont les suivantes :

*Eclipse JDT* La bibliothèque JDT, faisant partie du projet Eclipse, est l'API de parsing de code Java la plus populaire. A partir de code source Java, JDT construit un arbre de syntaxe abstrait (AST) qui peut ensuite être parcouru (en suivant le pattern Visitor) afin d'effectuer certaines actions sur les nœuds correspondant à des instructions qui nous intéressent. Cette bibliothèque existe depuis longtemps, est très activement maintenue par bon nombre de développeurs et possède un support complet des spécifications de toutes les versions du langage Java (de 1 à 11). Elle présente cependant deux points faibles : sa difficulté à mettre en place et sa relative lenteur. En effet, afin d'utiliser des fonctionnalités avancées de JDT, il faut faire appel à plusieurs autres API du projet Eclipse. De plus, de par sa grande complexité en terme de fonctionnalités offertes, JDT prend du temps et de la mémoire pour parser un grand nombre de fichiers source et à construire leur AST quand on veut résoudre tous les liens existant entre les symboles (par exemple entre l'instanciation d'une classe et sa déclaration).

*JavaParser*[7] JavaParser est une autre API de parsing de code source Java, la deuxième plus populaire après JDT. Elle est également activement maintenue, supporte toutes les versions de Java de 1 à 11 et a un fonctionnement similaire à JDT : elle construit un AST à partir du code source Java et permet le parcours de celui-ci via une implémentation du pattern Visitor. La construction des AST est plus rapide qu'avec JDT, mais JavaParser semble ne pas pouvoir résoudre tous les liens dans le code. Son utilisation est cependant plus aisée car il n'est pas nécessaire de faire appel à d'autres API pour accéder aux fonctionnalités plus avancées.

**Remarques annexes** Il existe quelques différences supplémentaires — bien que moins importantes que les points précédents — qui méritent d'être exposées :

— **Complexité en temps** Quel que soit l'outil utilisé, il est plus rapide d'effectuer une analyse de bytecode qu'une analyse de code source. Cela est dû au fait que beaucoup de liens sont résolus à la compilation et ne nécessitent pas d'être recalculés

7. `https://javaparser.org/`

(par exemple, lors de l'instanciation d'une classe, on connaît directement son nom complet) là où les parseurs de code source doivent les construire.

— **Disponibilité des fichiers** Bien que cela survienne très rarement, il est possible que l'utilisateur veuille analyser l'architecture d'une application dont il ne possède que les fichiers de code compilé. Dans ce cas, l'analyse de code source ne peut se faire (où alors il faut décompiler le code, procédé assez délicat) et il faut impérativement utiliser l'analyse de code compilé. Dans le cas contraire (le plus courant), on ne possède que le code source. On peut alors obtenir le bytecode aisément en compilant le programme. Il est donc possible d'utiliser autant l'analyse de code source que l'analyse de bytecode.

— **Informations divergentes** Au cours de la compilation, certaines portions du programme sont transformées. Parmi ces modifications, nous avons repéré (mais il en existe potentiellement d'autres) :
  — l'ajout de deux méthodes dans les classes d'énumération : `values` et `valueOf`.
  — l'ajout d'un attribut dans les classes internes qui référence leur classe englobante.
  — l'ajout d'un constructeur par défaut si celui-ci n'est pas spécifié.

  Ceci implique que le code compilé possède une structure légèrement différente du code source. L'analyse de bytecode détecte donc des dépendances et méthodes que l'analyse de code source n'est pas en mesure de récupérer.

### 4.3.2. Choix décisifs et limitations

**Conservation des deux implémentations d'analyse** Bien que l'analyse de code source s'avère être la plus efficace, nous avons décidé de conserver également notre parseur de code compilé. En effet, ceci permet de mener des expériences comparatives entre ces deux méthodes, ce qui constitue un des besoins principaux du projet. Cependant, il pourrait être opportun de supprimer l'analyse de bytecode car il est compliqué de rester cohérent dans la gestion des spécificités des deux parseurs (cf. 4.3.1).

**Performance et optimisation** L'analyse du code s'effectue en deux temps : analyse du niveau d'abstraction et analyse des dépendances. Cette séparation induit une complexité en temps plus importante car chaque fichier est parsé deux fois. De plus, l'analyse étant l'étape la plus longue, l'impact sur la durée d'exécution est non négligeable. Cependant, les performances ayant peu d'importance pour le client, nous avons fait le choix de privilégier une architecture plus propre en conservant la distinction entre les deux analyses. Les différents tests que nous avons effectués montrent de bonnes performances, y compris sur des projets de très grande taille (cf. 5.3).

**Gestion de la multiplicité des dépendances** Nous ne considérons qu'une seule et unique dépendance par type et par granule. Par exemple s'il y a dix dépendances d'utilisation d'un granule A vers un granule B, on n'en considère qu'une seule. Certaines interprétations de la métrique de Martin tiennent compte de la multiplicité des dépen-

dances[**?** ], d'autres non[**?** ]. Les deux cas semblent donner des résultats satisfaisants. Nous avons pris le parti de ne pas exploiter cette information par simplicité d'implémentation.

**Dépendances externes** Nous ne considérons les dépendances qu'au sein d'un même projet. En effet, bien qu'il soit facile de récupérer les dépendances efférentes vers des granules externes, l'inverse ne peut pas être exhaustif. Ce choix a donc pour but d'éviter d'extraire un nombre disproportionné de dépendances efférentes par rapport aux dépendances afférentes, ce qui perturberait les calculs des métriques. De ce fait les dépendances vers des bibliothèques externes, par exemples l'API standard, sont ignorées.

**Éléments ignorés** Nous avons fait le choix d'ignorer certains éléments lors des analyses, pour diverses raisons. En voici la liste :

- **Les constructeurs** : Lors du comptage du nombre de méthodes d'une classe, les constructeurs sont ignorés. Ce choix est motivé par le fait qu'à la compilation, un constructeur par défaut est ajouté aux classes n'en possédant pas. Ceci entraîne des incohérences entre les mesures effectuées sur le code source et le code compilé.
- **Les classes internes et anonymes** : A la demande du client et afin de simplifier l'analyse, les classes internes et anonymes sont ignorées par les deux analyses.

**Classes et fichiers** : L'application considère que chaque fichier contient une unique classe portant le même nom que celui-ci. De ce fait, toute autre classe est ignorée.

# 5. Tests

Nous avons — tout au long de notre projet — développé une série de tests permettant de contrôler le bon fonctionnement de notre implémentation. Les outils et méthodes que nous avons utilisés ayant évolué au cours de la période de développement, ces tests nous ont permis d'identifier les problèmes de régression. Enfin, au travers de la mise en œuvre des tests directement liés à notre domaine d'étude (analyse et métrique), nous avons pu appréhender les spécificités et ainsi améliorer le cœur de notre programme.

Pour la mise en oeuvre de ces tests, nous avons utilisé le framework **JUnit 5**.

Les composants de notre application ayant été soumis à une série de tests sont les suivants :

— L'exploration du système de fichier et la création de structure représentative du projet.
— Les métriques et leurs différents calculs attachés.
— Le graphe, ses primitives ainsi que les services qui y sont associés.
— L'analyse de code.

## 5.1. Infrastructure de test

**Vérité terrain** Le package `analysis` ne pouvant pas être testé de manière triviale, nous avons mis en oeuvre une **vérité terrain annotée** qui nous a permis de vérifier la bonne concordance des données de dépendances et des données d'abstraction extraites par notre application avec les données réellement présentes dans un projet Java.

Cette vérité terrain est constituée d'un ensemble de projet Java présentant des cas divers de dépendances et d'abstraction. Nous avons mis en place un système d'annotation Java permettant de simplifier la saisie et la création d'éléments de cette vérité terrain.

Listing 1 – Exemple de classe annotée

```
@ClassInfo(
    numberOfMethod = 1,
    numberOfAbstractMethod = 0,
    Ca = 2,
    Ce = 3,
    I = 0.6,
    A = 0,
    Dn = 0.4
)
@Dependency(dependencyTo = A.class, type = DependencyType.Inheritance)
public class B extends A {
    @Dependency(dependencyTo = C.class, type = DependencyType.Aggregation)
    private C c;
    @Dependency(dependencyTo = D.class, type = DependencyType.UseLink)
    public void doSomething() { D d = new D(); }
}
```

Une classe `GroundTruthManager` a pour but de charger en mémoire les différents projets composant la vérité terrain et offre des méthodes d'accès à ces différentes données. Le code des tests de nos parseurs est ainsi simplifié.

## 5.2. Tests unitaires et d'intégration

Le développement de tests unitaires s'est révélé compliqué à cause de notre pipeline de traitement. En effet, certains packages de notre application ne peuvent pas fonctionner seuls et nécessitent d'être intégrés dans leur contexte de fonctionnement. Nous avons tout de même réalisé des tests qui vérifient le fonctionnement correct de notre implémentation pour chacun des packages de notre projet (excepté `presentation`).

**Package project** Obtenir une bonne représentation structurelle d'un projet est l'un des points principaux de notre implémentation. Dans cette mesure, nous testons cette fonctionnalité présente dans le package `project` pour les deux types d'exploration de fichiers (sources et compilés).

**Package analysis** L'analyse de code constituant le coeur de notre projet, nous testons les parseurs de couplage et de niveau d'abstraction pour les deux méthodes d'analyse (code source et bytecode). Pour ce faire, nous parcourons l'ensemble des projets de la vérité terrain en vérifiant la conformité des données des annotations avec les données extraites.

**Package metrics** Au niveau du package `metrics`, nous avons testé en premier lieu la validité du calcul de métriques en fournissant des données incohérentes et en vérifiant la levée d'exception correspondante. Nous avons, dans un second temps, annoté chaque classe avec les valeurs attendues pour les métriques (`ClassInfo`). Nous avons ensuite vérifié la correspondance entre ces données et celles réellement calculées par l'application.

La comparaison à l'échelle des packages est moins évidente. Nous avons, comme pour les classes, envisagé d'utiliser des annotations, appliquées cette fois sur le fichier `package-info`. Cependant, l'extraction de ces annotations s'est avérée plus complexe. En effet, les fichiers `package-info`, tout comme les fichiers `module-info`, ne sont pas compilés. Ceci rend la lecture des annotations impossible avec notre implémentation actuelle (lecture par introspection).

**Package graph** Nous avons également testé que la structure de graphe que nous proposons est correcte. Pour ce faire, nous avons repris les primitives essentielles qui sont utilisées dans le cadre du calcul des métriques. Nous avons également testé le service de création de graphe et sa bonne intégration dans le pipeline de notre projet.

## 5.3. Tests de performance

Nous avons effectué des tests de performance. Ces derniers ont consisté en l'exécution de l'application sur des projets Java plus ou moins large (dont notamment l'API Java elle-même). Ces tests nous ont permis d'obtenir un tableau représentatif des performances de notre application.

Les tests de performance ont été effectués sur la configuration suivante :

— **Java** : Oracle JRE 1.8 revision 201 (Win x64)
— **Système d'exploitation** : Windows 10 v. 1809
— **Processeur** : i7 8700K 4.7Ghz
— **RAM** : 16Go DDR4

| Projet | Nombre de classes | Nombre de packages | Temps d'exécution |
|---|---|---|---|
| Oracle JDK 1.8 | 7651 | 486 | 10 min 29 s |
| JUnit 5 | 435 | 58 | 0 min 15 sec |
| JMetrics | 41 | 6 | 0 min 2 sec |

## 5.4. Tests de fonctionnement

**Comparaison avec JDepend** Afin de situer notre implémentation de la métrique de Martin par rapport à une autre déjà existante (cf. 2.4.2), nous avons analysé plusieurs codes avec notre application et JDepend en parallèle. Il en est ressorti que les valeurs des métriques calculées par les deux programmes étaient très éloignées. Les raisons de ce problème sont diverses :

— JDepend effectue une analyse sur le bytecode (à l'aide d'un parseur écrit *from scratch*). Il est plus complet que notre parseur de bytecode utilisant l'introspection mais moins que notre parseur de code source (à cause des limitations du bytecode, cf. 4.3.1). Que ce soit avec l'une ou l'autre des méthodes de notre application, les ensembles de dépendances extraits sont alors trop différents pour être comparés.
— JDepend tient compte des dépendances vers des bibliothèques externes dans le calcul de $Ce$, mais pas dans celui de $Ca$ (en effet, il est techniquement impossible de connaître tous les granules existants en dehors d'un projet qui dépendent d'un granule en particulier). Ceci induit une certaine incohérence dans les valeurs de couplage, avec un $Ce$ souvent très élevé par rapport au $Ca$. De ce fait, la plupart des granules ont une instabilité élevée. Les granules très concrets ont alors une distance à la séquence principale très faible la plupart du temps. En outre, étant donné que les granules concrets sont une majorité dans bon nombre de projets (il y a souvent plus d'une implémentation pour chaque abstraction), JDepend affiche une distance moyenne très faible sur tous les codes analysés.
  De notre côté, nous avons choisi d'ignorer entièrement les dépendances externes. La conséquence de cette décision est une valeur de $Ce$ moins élevée, entraînant une valeur d'instabilité réduite. En résulte une distance plus grande que celle rapportée par JDepend.

Pour ces raisons, nous n'avons pas jugé pertinentes les expérimentations menées avec JDepend.

# 6. Résultats & Interprétations

**Données extraites** A l'exécution de notre application, plusieurs fichiers sont générés (au format CSV et DOT) permettant d'exposer les données de l'analyse à l'utilisateur. Ces données sont les mesures de dépendances sous forme de graphe (représentation graphique ou matricielle) ou de liste, et les données de métriques. Ces données sont classées par niveau de granularité (*class scale* et *package scale*).

## 6.1. Présentation

**Représentation proposées** Afin d'offrir à l'utilisateur des outils d'interprétation des données brutes extraites par notre analyse, nous avons mis en place une série de scripts Python permettant la lecture et l'étude des fichiers csv générés. Les différentes représentations proposées sont les suivantes (les plots présentés expose les données de l'analyse de notre propre programme au niveau de granularité des packages) :

— **Histogramme (Métriques)** : Cette première représentation est la plus simple. Elle nous permet d'étudier sous la forme d'histogramme les différentes composantes de la métrique.

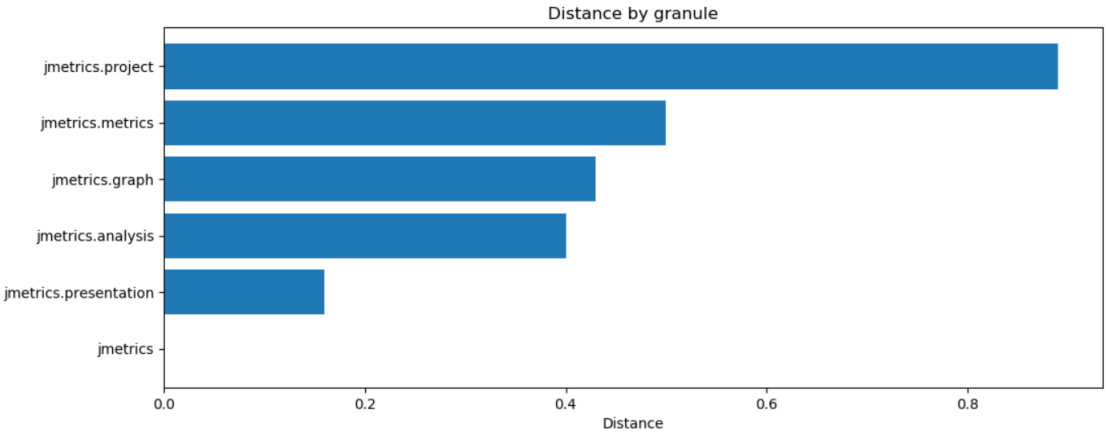


FIGURE 10 – Histogramme de la composante distance

— **Histogramme multicolore (Dépendances)** : Cette représentation nous permet d'étudier le poids des types de dépendances dans les composantes de couplages. Ainsi, cette représentation permet une étude de l'impact des types de dépendances (et de manière indirect, de multiplicité) sur la valeur de stabilité. Elle étudie de manière disjointe les données de couplage afférent et efférent (un seul axe étant disponible pour l'affichage de valeur). Nous étudions les données de couplages de manière conjointe depuis une autre représentation (Représentation 2D) décrite ci-dessous.

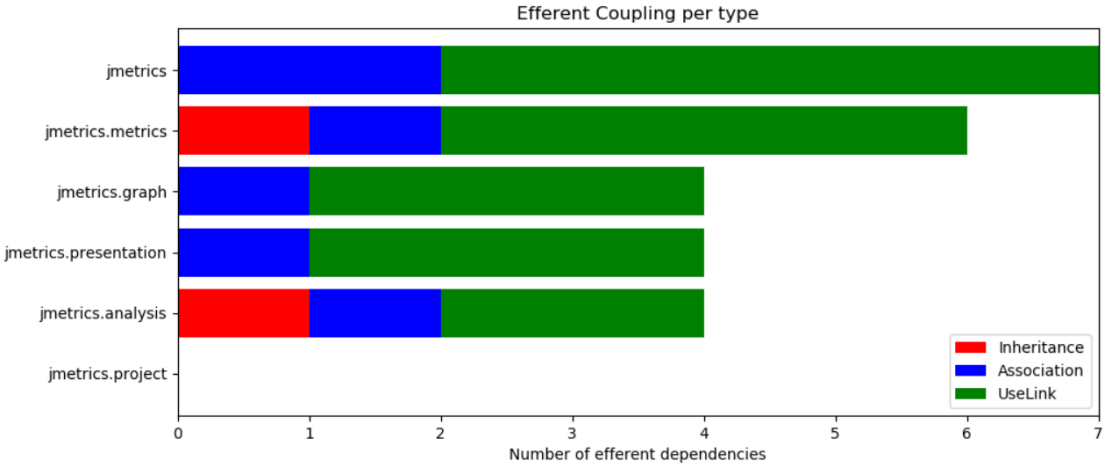


FIGURE 11 – Histogramme du couplage efférent

— **Représentation 2D (Instabilité et Abstraction)** : Cette représentation met en relation les données d'instabilité et d'abstraction sur un axe orthogonal. Cette représentation permet d'afficher les différents granules en fonction de leur stabilité et de leur niveau d'abstraction. A l'image de ce que décrit Martin dans son article, nous positionnons sur le plan les zones de souffrance et d'inutilité ainsi que la séquence principale. Cette représentation graphique, bien que soumise à certaines critiques est très représentative de la métrique énoncée par Martin.
En survolant un point du plan, il est possible d'obtenir les coordonnées du plan (données d'abstraction et de stabilité), la distance du point à la séquence principale et la liste des granules présents sur la position.

— **Représentation 2D (Couplage afférent et efférent)** : Cette représentation met en relation les données de couplages afférent et efférent sur un axe orthogonal. Cette représentation nous permet d'étudier la stabilité en séparant la propriété de responsabilité et d'indépendance. De plus, elle nous permet d'étudier les extrema des valeurs de couplage. Nous faisons ici abstraction de la donnée de type des dépendances pour nous concentrer sur la valeur de couplage uniquement.

— **Analyse statistique des métriques** : Enfin, un dernier script a était mise en place dans le but d'étudier des fonctions statistiques appliquées au composantes de métriques. Ce script permet d'afficher sur la sortie standard les valeurs de moyennes, médiane, variance et de déviation standard à l'ensemble des composantes de la métriques.

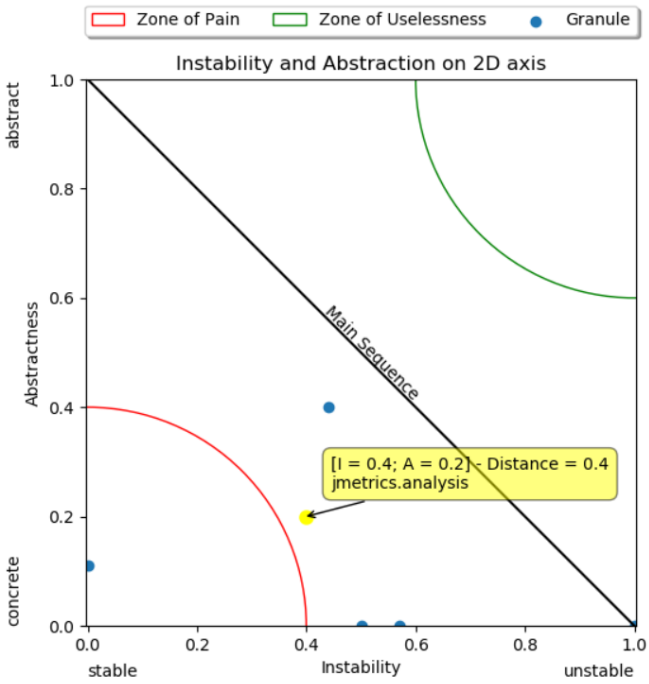


FIGURE 12 – Positionnement des granules sur la Main Sequence

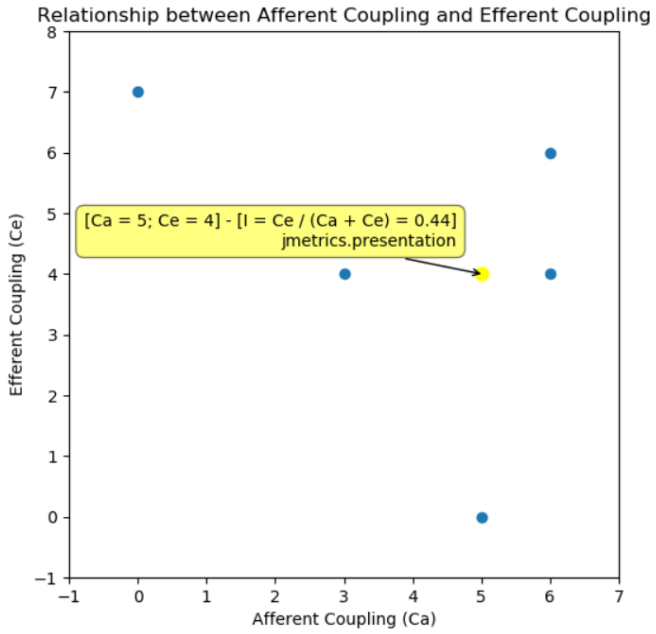


FIGURE 13 – Couplage afférent et efférent sur axe orthogonal

## 6.2. Expérimentations

**Complete**

### 6.2.1. Analyse des (GoF's) Design Pattern

Nous avons — conjointement à l'unité d'enseignement PDP — une unité d'enseignement Architecture logicielle dans laquelle nous découvrons les Design Patterns exposés par les auteurs du *Gang of Four*[refBib]. Dans ce contexte, nous avons souhaiter créer une passerelle entre ces deux unités d'enseignement en procédant à l'étude de l'analyse de différentes implémentations de design pattern par notre programme.

La motivation ayant mené à l'élaboration des design patterns étant la production d'architecture flexible, extensible, maintenable et réutilisable, leur étude au travers de la métrique de Martin semble être intéressant. De plus, nous pouvons aisément trouver sur internet, des catalogues de petit programmes mettant en oeuvre les différents patterns ce qui est un avantage supplémentaire. Dans cette étude, nous nous baserons sur le repository **Refactoring_Guru**[8] de l'auteur *Alexey Pyltsyn*.

Nous nous intéresserons ici a des implémentations dans leurs formes minimales. Considérer de tels implémentations nous permet d'obtenir une forte **reproductibilité**. En effet, dans leurs versions minimales, les implémentation de pattern possèdent toutes des valeurs de métriques presque similaire.

Nous noterons cependant que, découpler de tout environnement, certaines classes peuvent avoir des valeurs de métriques qui ne correspondent pas à leurs valeurs dans une utilisation normal du pattern. Par exemple, une classe Singleton, ou encore une classe Component (classe mère d'une instance de pattern composite) aura tendance à avoir un couplage afférent important qui n'apparaîtra pas dans les exemples. Nous nous permettrons ainsi de faire varier certaines des valeurs analysées.

Enfin, nous noterons — comme l'indique les auteurs du GoF — qu'il est tout à fait possible de faire collaborer plusieurs patterns. Nous tenterons d'étudier un cas de collaboration.

Ci-dessous nous avons les valeurs moyennes de distance pour chacun des patterns :

8. `https://github.com/RefactoringGuru/design-patterns-java`

| Pattern | Valeur moyenne de Dn |
|---|---|
| abstract_factory | 0.26 |
| adapter | 0.55 |
| bridge | 0.25 |
| builder | 0.51 |
| chain_of_responsibility | 0.37 |
| command | 0.32 |
| composite | 0.32 |
| decorator | 0.28 |
| facade | 0.5 |
| factory_method | 0.26 |
| flyweight | 0.41 |
| iterator | 0.28 |
| mediator | 0.34 |
| memento | 0.33 |
| observer | 0.32 |
| prototype | 0.29 |
| proxy | 0.34 |
| singleton | 0.8 |
| state | 0.29 |
| strategy | 0.4 |
| template_method | 0.23 |
| visitor | 0.37 |

**Descriptions associées** Nous allons maintenant fournir quelques explications pour certain patterns :

- **Abstract Factory (mean(Dn) = 0.26)** :
  - **AbstractProduct** : Les produit abstrait ont un niveau d'abstraction à 1 ; un Ca fort, un Ce nul, et par conséquent une instabilité à 0. Il se situe donc sur la main sequence.
  - **AbstractFactory** : La fabrique abstraite est totalement abstraite. Elle possède

### 6.2.2. Étude de refactoring

**Visualisation de Martin sur l'évolution de D selon les releases : cf. p.267 Book**

# 7. Conclusion et perspective

## 7.1. Bilan

**Difficultés rencontrées** La compréhension du domaine, préambule à l'élaboration du projet, a constitué une grande difficulté. En effet, des concepts tels que la stabilité sont longtemps restés peu clair. Nous avons ainsi dû retravailler ces différentes notions théoriques à de multiples reprises.

Au début du projet, sans aucune expérience dans l'analyse de code, la comparaison des méthodes d'analyse fut une tâche complexe. Nous avons ainsi passé beaucoup de temps sur l'étude de ces différents outils permettant ces analyses.

**Récapitulatif des besoins** Nous avons réussi à prendre en satisfaire l'ensemble des besoins exposés par le client. Nous exposons tout de même une série d'améliorations dans la section suivante.

## 7.2. Perspectives

Ce projet est, comme tout projet, améliorable. Nous avons présenté dans la partie limitation (cf. 4.3.2) une liste de points sur lesquels nous pourrions revenir.

En plus de celle-ci, nous avons pensé à la mise en oeuvre de quelques améliorations :

- **Optimisation mémoire** : Actuellement, notre représentation des granules après analyse conserve une référence vers leur fichier/dossier. Or, d'après le pipeline, il n'est plus nécessaire d'accéder à nouveau aux fichiers après l'analyse. De ce fait, nous pourrions corriger l'implémentation afin de supprimer cette référence inutile.
- **Formats de sortie** : Les sorties de notre programme sont actuellement formatées en CSV. Ce format est pratique pour représenter une série de données partageant la même structure. En conséquence, il est nécessaire de créer un fichier par type de données à représenter (valeurs de métriques, dépendances, ...). Un format plus riche, comme le XML, pourrait permettre de représenter plus d'informations dans moins de fichiers, tout en gardant la facilité de lecture du CSV.
- **Visualisation de données** : En plus des différentes visualisations de données proposées, il serait intéressant de penser et de mettre en oeuvre d'autres représentations permettant une bonne interprétation des données.
- **Niveaux de granularité** : Il serait intéressant de considérer de nouveaux niveaux de granularité tels que les modules, ainsi que les attributs et méthodes au sein d'une classe.
- **Améliorer le calcul de métrique** : La bibliographie sur la métrique de Martin étant très vaste, il est toujours possible d'améliorer les différentes spécificités du calcul de celle-ci.
- **Application du SDP** : Comme abordé dans la description du domaine, un calcul itératif sur le graphe permettrait de minimiser le poids des dépendances stables dans

le calcul du couplage en propageant la stabilité. Il serait alors possible d'appliquer le SDP, qui n'est actuellement pas pris en compte dans la métrique.

— **Métriques supplémentaires** : Grâce à la flexibilité de notre package `metrics`, l'intégration de métriques est facilitée. On pourrait donc envisager l'ajout de métriques supplémentaires (les métriques de Chidamber et Kemerer, ...).

— **Intégration aux IDE** : Afin de faciliter l'utilisation de notre application, il pourrait être opportun de l'intégrer aux IDE les plus utilisés (Eclipse, IntelliJ, ...).

— **Gestion des releases d'un projet** : Pour voir l'évolution des métriques en fonctions des releases (fonctionnalités apportées et *refactor* effectués), il pourrait être envisagé d'intégrer notre application à des outils de gestion de version (Gitlab, ...).

# Annexes

## A. Analyse de projet : Exemple

Dans le but de vérifier que l'application calcule les valeurs de métrique correctement, il est nécessaire d'écrire de petits logiciels simples présentant des configurations de dépendances différentes. Nous donnons ici un exemple élémentaire de projet ainsi que les sorties de l'application pour celui-ci.

Listing 2 – Classe Vehicle

```
public abstract class Vehicle {
    private int nbWheel;
    private ArrayList<Wheel> wheels;
    public abstract void move();
    public void setNbWheel(int i) { }
    public void addWheel(Wheel w) { }
}
```

Listing 3 – Enumeration Material

```
public enum Material {
    Plastic,
    Metal,
    Carbon
}
```

Listing 4 – Classe Airplane

```
public class Airplane extends Vehicle {
    @Override
    public void move() { /* ... */ }
}
```

Listing 5 – Classe Car

```
public class Car extends Vehicle {
    @Override
    public void move() { /* ... */ }
}
```

Listing 6 – Classe Wheel

```
public class Wheel {
    private Material material;
    public Material getMaterial() { return this.material; }
    public void setMaterial(Material m) { this.material = m; }
}
```

Table 1 – Tableau exposant les métriques

| Granule | Ca | Ce | I | A | Dn |
|---|---|---|---|---|---|
| **Vehicle** | 2 | 2 | 0.5 | 0.33 | 0.17 |
| **Material** | 2 | 0 | 0 | 0 | 1 |
| **Airplane** | 0 | 1 | 1 | 0 | 0 |
| **Car** | 0 | 1 | 1 | 0 | 0 |
| **Wheel** | 2 | 2 | 0.5 | 0 | 0.5 |

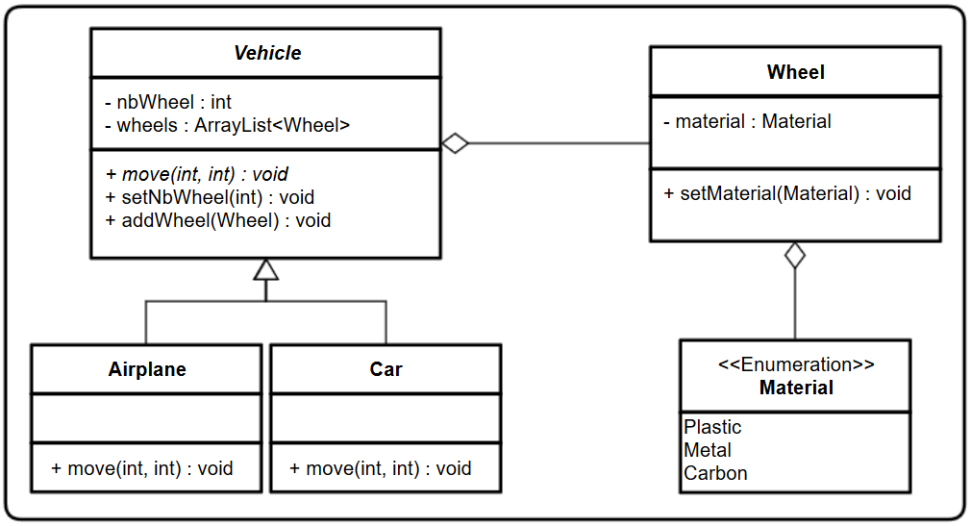


FIGURE 14 – Diagramme de classe du projet exemple

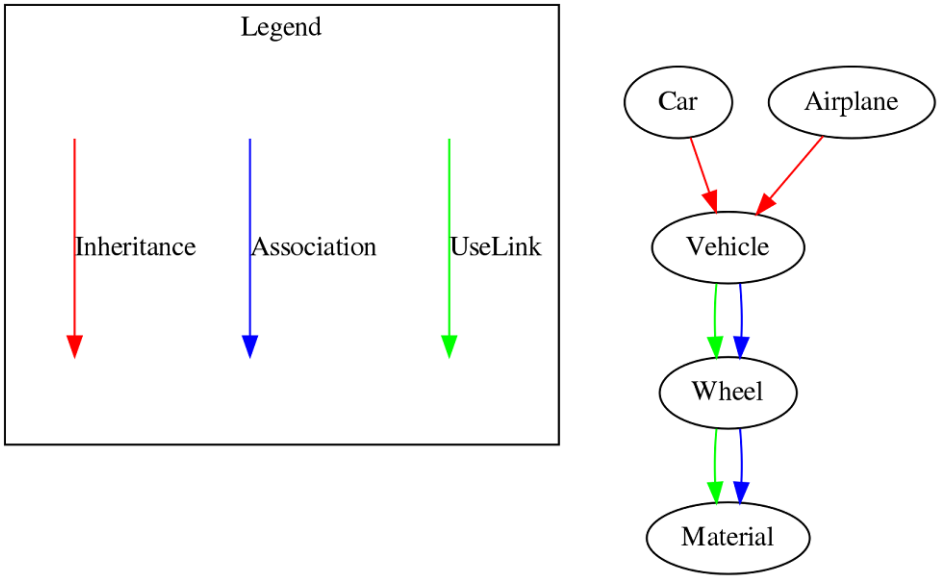


FIGURE 15 – Graphe des dépendances du projet exemple construit par l'application