université de BORDEAUX

PdP - Métriques de Maintenabilité

Cahier d'analyse des besoins

Vendredi 1 Février 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

A une époque où certaines applications peuvent être distribuées sur des milliards de machines, la maintenabilité est devenue un enjeu majeur dans le domaine du développement logiciel.

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 définit par Martin[1] 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 de paradigme Orienté Objet.

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 travers de métriques logicielles afin d'obtenir des informations sur la maintenabilité en sortie.

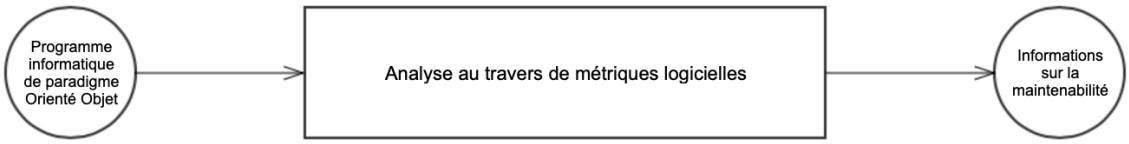


FIGURE 1 – Vision simplifié du fonctionnement de l'application

2 Description du domaine

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. Un bon moyen de mesurer une telle chose est d'appliquer le calcul d'une métrique logicielle. Bien que ces métriques soient nombreuses, l'application s'appuiera sur celle définie par Robert Martin[1]. Cette section a pour vocation de décrire cette métrique ainsi que tout le vocabulaire nécessaire à sa compréhension.

2.2 Terminologie

2.2.1 Dépendances

Définition Une classe (ou ensemble de classes) A *dépend* d'une classe (ou ensemble de classes) B dans le cas où A a besoin de B pour être compilé. Cette *relation de dépendance* peut s'exprimer sous différentes formes :

- **Dépendance par composition** : A (ou une classe de A) possède un attribut de type B (ou d'une classe de A).
- **Dépendance par application** : A (ou une classe de A) appelle une méthode de B (ou d'une classe de B).
- **Dépendance par héritage / par implémentation** : A (ou une classe de A) est de type B (ou du type d'une classe de B).
- **Dépendance par transitivité** : A dépend de C qui dépend de B.

Couplage Le degré de couplage est une mesure de l'*interdépendance* entre les différents sous-systèmes d'un programme. On parle de couplage fort pour signifier que l'interdépendance est élevée. On peut étudier cette interdépendance à plusieurs échelles (classes ou ensembles de classes).

Cohésion La cohésion représente le degré de liaison et de collaboration des éléments appartenant à un même composant. 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. A l'inverse, une faible cohésion indique que le composant effectue plusieurs actions sans rapport entre-elles et pourrait certainement être divisé en sous-parties.

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[1]) :

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 casser 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 subit la modification. De plus, la correction de ces erreurs amène souvent à davantage de nouveaux problèmes. Il devient alors difficile d'estimer la qualité du système.

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 tout 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.

Stabilité La stabilité désigne la capacité d'un composant à ne pas varier dans le temps. Moins un composant a de risques de changer, plus il est stable. Un composant peut être stable soit parce qu'il n'a aucune dépendance et donc aucune raison de s'adapter à d'autres changements, soit parce que beaucoup d'autres composants dépendent de lui et qu'un changement aurait trop de répercussions sur ceux-ci. Un composant très stable est un composant qui combine les deux propriétés précédentes, à savoir aucune dépendance et beaucoup de composants dépendant de lui.

Responsabilité Un composant responsable est un composant dont dépendent de nombreux autres. Un tel composant ne doit pas être souvent modifié car chaque changement peut causer de nombreux problèmes, compte tenu du nombre de composants dépendant de lui. La responsabilité joue en la faveur de la stabilité.

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.

Remarque sur les dépendances Les dépendances d'un composant peuvent être classées en deux types opposés : les bonnes dépendances (Good dependencies) et les mauvaises dépendances (Bad dependencies). Une bonne dépendance 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 est une dépendance dont la cible est instable. Ce sont, assez naturellement, ces dépendances qu'il faut éviter.

2.2.4 Catégories de classes ("Class categories")

Dans l'article principal de l'étude[1], R. Martin explique que la métrique qu'il définit a pour but d'étudier les dépendances entres les différentes catégories de classe (*class categories*) d'un programme. Il motive ce choix en précisant que « *dans une même catégorie, les classes sont censées être très interdépendantes* ».

Définition Afin d'appréhender plus précisément le concept de catégorie de classes utilisé par Robert C. Martin, il est utile de se référer à la définition énoncée par son créateur Grady Booch[2]. Booch a mis au point ce concept dans le but de faciliter la mise en oeuvre de représentations structurelles d'applications dans la méthode qui porte son nom. Il définit ce concept de la manière suivante :

> [A class category is] A logical collection of classes, some of which are visible to other class categories, and others of which are hidden. The classes in a class category collaborate to provide a set of services.
>
> — G. Booch - [2] (p.253)

Règles d'appartenance à une catégorie Afin de préciser le concept de catégorie, Martin liste un ensemble de règles (énoncées par ordre d'importance) définissant l'appartenance d'une classe à une catégorie :

1. Les classes appartenant à une même catégorie sont fermées entre elles contre toute force de changement (au sens du principe ouvert/fermé). Cela signifie que si une classe appartenant à une catégorie doit être changée, l'ensemble des classes de cette même catégorie devra changer pour s'adapter.
2. Les classes appartenant à une même catégorie ont tendance à être réutilisées ensemble (cette propriété étant une conséquence de leur forte interdépendance).
3. Les classes appartenant à une même catégorie partagent un but commun.

Problématique de cette notion Il semble complexe de définir un outil qui puisse déterminer de manière autonome l'appartenance d'une classe à une catégorie (au sens de Booch). En effet, la cohésion et l'interdépendance entre les différentes classes d'une catégorie ne décrivent pas forcément la volonté du développeur de créer un ensemble de classes cohérent : cela peut être le reflet d'un code mal structuré (couplage fort).

La détermination de catégorie dans une première analyse de logiciel devrait donc se baser sur la structuration qu'offrent des concepts propres au langage (namespace C++, package Java, etc.).

2.3 Métrique de Martin

2.3.1 Présentation et définition

La métrique de Martin a été définie pour la première fois en 1994 par Robert Martin[1]. L'auteur l'a par la suite citée au sein d'autres ouvrages[3], et une large bibliographie scientifique mentionne, critique et complète celle-ci[4][5][6][7][8]. La métrique s'articule autour de 2 notions centrales : la stabilité et le niveau d'abstraction (cf. 2.2.3).

Définitions Martin présente une métrique principale : la distance entre une *catégorie* représentée par un point de coordonnées (Instabilité, Abstraction) et la "Main Sequence", une droite représentant le positionnement idéal des catégories. Plus cette distance est grande, moins la classe correspond au pattern recherché. Afin de calculer cette distance, il est nécessaire de calculer plusieurs autres métriques. Voici une liste de celles-ci :

- **Afferent Coupling, noté Ca** (Le couplage afférent, couplage "entrant") : Il s'agit du nombre de classes externes qui dépendent d'une (ou plusieurs) classes dans la catégorie.
- **Efferent Coupling, noté Ce** (Le couplage efférent, couplage "sortant") : Il s'agit du nombre de classes de la catégorie qui dépendent d'une classe externe.
- **Instability, notée I** (L'instabilité) : Il s'agit d'une quantification de l'instabilité d'une catégorie (cf. 2.2 pour une définition de la stabilité). Cette métrique fait intervenir les deux précédentes.

— **Abstractness, notée A** (Le niveau d'abstraction) : Il s'agit d'une quantification du niveau d'abstraction d'une catégorie.
— **Distance, notée D (ou Dn pour la version normalisée)** (La distance par rapport à la Séquence Principale, définie ci-dessous.) : Il s'agit d'une mesure de la distance perpendiculaire d'une catégorie à la Séquence Principale. Cette distance donne une idée de la qualité d'une catégorie : le but est de minimiser cette valeur.

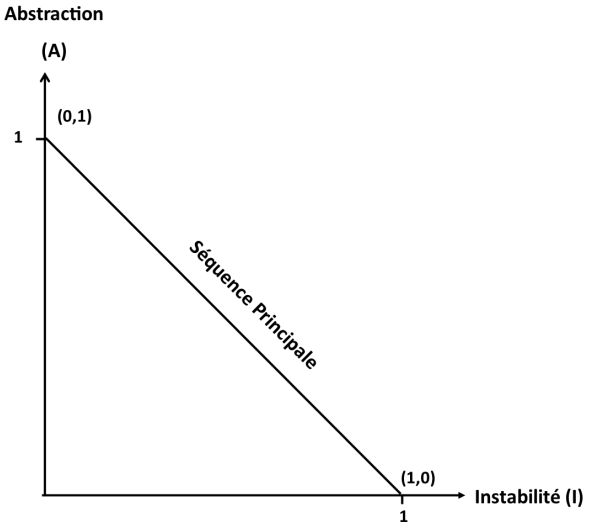


FIGURE 2 – Séquence principale

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, une catégorie est représentée 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'une catégorie instable mais entièrement concrète, et (1,0), qui représente une catégorie très stable et entièrement abstraite. Des compromis sont également possibles : il s'agit de la Séquence Principale, une droite passant par (0,1) et (1,0). Les catégories proches de cette droite sont considérées comme bien équilibrées. Dès lors, une faible distance D sera indicatrice de catégorie bien formée alors qu'une distance important indiquera que la catégorie est potentiellement améliorable.

2.3.2 Formalisme

Variables Soit **Π** un projet de développement. Un projet **Π** est composé par un ensemble de n ($n > 0$) classes et de m ($m \geq 0$) packages. On note **U**, l'univers des classes. $\mathbf{U} = \{c_1, c_2, \ldots, c_n\}$ où $c_1, c_2, ...c_n$ représentent les n classes de **Π**.

Une catégorie **C** est un sous-ensemble non-vide de U : $\mathbf{C} = \{c_i, \ldots, c_j\} \subseteq \mathbf{U}$, avec $0 > i, j \geq n$.

On définit **P** comme l'ensemble des **C**. On appelle catégorie élémentaire une catégorie de cardinal 1 (c.à.d. une catégorie composée d'une unique classe).

Fonctions On note les fonctions suivantes :
— **DaC**(**C**) = Le nombre de classes dépendant d'une catégorie **C**.
— **DeC**(**C**, **P**) = Le nombre de classes dont dépend une catégorie **C** dans son univers **P**.
— **NoM**(**C**) = Le nombre de méthodes présentes au sein d'une catégorie **C**.
— **NoAM**(**C**) = Le nombre de méthodes abstraites présentes au sein d'une catégorie **C**.

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

— $\mathrm{Ca}(\mathbf{C}) = \mathbf{DaC}(\mathbf{C}) \in \mathbb{N}$
— $\mathrm{Ce}(\mathbf{C}, \mathbf{P}) = \mathbf{DeC}(\mathbf{C}, \mathbf{P}) \in \mathbb{N}$
— $\mathrm{A}(\mathbf{C}) = \mathbf{NoAM}(\mathbf{C}) \ / \ \mathbf{NoM}(\mathbf{C}) \in |0, 1|$
— $\mathrm{I}(\mathbf{C}) = \mathrm{Ce}(\mathbf{C}) \ / \ (\mathrm{Ca}(\mathbf{C}) + \mathrm{Ce}(\mathbf{C}) \in |0, 1|$
— $\mathbf{D(C) = \ | \ (A + I - 1) \ / \ \sqrt{2} \ | \ } \in |0, 0.707|$

La distance peut se normaliser dans un intervalle de 0 à 1 en utilisant une formule simplifiée :

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

Finalité du calcul Finalement, on remarque que toutes les métriques participent à calculer la distance à la Séquence Principale. Une fois celle-ci calculée pour toutes les catégories, il est facile de repérer celles auxquelles il semble nécessaire de prêter attention.

2.4 Intérêts

Comprendre la métrique Un intérêt annexe à l'objectif principal de l'application est que son développement est un moyen efficace de comprendre et de mener une réflexion sur la métrique. Ceci permettrait de l'évaluer dans son ensemble et d'apporter des améliorations.

Critique et pistes d'amélioration Le problème majeur que l'on peut reprocher à la métrique de Martin est que le résultat du calcul des métriques ne constituent pas des références absolues, ce sont des indicateurs qui peuvent servir à détecter les zones de code auxquelles prêter une attention particulière.

Les métriques décrites par Chidamber et Kemerer[9] définissent un indicateur plus simplifié que les métriques de Martin. En effet, ils définissent six métriques telles que la profondeur de l'arbre d'héritage (DIT), le nombre de fils d'une classe (NOC), le manque de cohésion des méthodes d'une classe (LCOM) et bien d'autres qui sont utiles dans l'étude du couplage et la cohésion au niveau de différentes classes. Certains éléments de ces métriques peuvent servir à améliorer la métrique de Martin.

> 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 [9] (p.229)

Martin présente sa métrique comme ne s'appliquant qu'aux catégories contenant plusieurs classes. Cependant, en utilisant le formalisme exprimé en section 2.3.2, la métrique de Martin peut s'appliquer à l'échelle des classes (catégories élémentaires).

Telle qu'elle est énoncée, la métrique de Martin considère les dépendances comme toutes égales. Or, une dépendance envers une catégorie stable ne devrait pas avoir le même poids qu'une dépendance envers une catégorie instable. Afin de raffiner le calcul des métriques, une analyse de la globalité de l'arbre 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 instable.

3 Application au langage Java

L'application aura pour but d'appliquer la métrique à des programmes écrits en langage Java. Dans ce cadre, une clarification de la manière dont certains concepts sont utilisables en Java est nécessaire.

3.1 Adaptation des notions

Catégories de classes La notion la plus proche de catégorie de classes dans le langage Java est celle de package. Cependant, il existe un glissement sémantique : bien que ce soit souvent le cas en pratique, un package Java n'est pas obligatoirement composé d'un ensemble de classes *collaborant* pour offrir un ensemble de services. Cette notion peut aussi s'appliquer aux modules.

3.2 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[9]). Pour le langage Java, on peut citer JHawk [1] en guise d'exemple. Certains de ces outils s'intègrent à des IDE comme Eclipse ou IntelliJ, ce qui facilite grandement leur mise en place et leur installation.

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 donc également analyser les archives jar).

A la suite d'une étude préliminaire du fonctionnement de JDepend, il ressort que celui-ci semble n'extraire que les informations de dépendance entre les packages (principalement au travers des déclarations *import* de Java). Il calcule et fournit alors les dépendances et valeurs de métriques à cette échelle. Il ne peut donc fournir aucune information sur les dépendances entre classes.

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

Remarque 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`

4 Expression des besoins

4.1 Besoins fonctionnels

Nous n'avons pas réalisé une étude complète [4] sur les outils et méthodes d'analyse à utiliser pour l'instant. Les deux choix possibles sont l'analyse de code source ou l'analyse de bytecode. Certains détails techniques n'ont pas encore été fixés et peuvent influencer les différents besoins énoncés ci-dessous.

Evaluer 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 la plus aisée et rapide à mettre en place. C'est cette solution qui sera retenue pour la suite du projet.

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 dans le corps des fonctions.

4.1.1 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.

→ **Test** : S'assurer que l'interface en ligne de commande accepte les chemins corrects (ceux qui mènent vers un répertoire local) et rejette les chemins incorrects (ceux qui mènent vers un répertoire inexistant).

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.

4. Une étude préliminaire est exposée en annexe

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.

4.1.2 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 de la métrique.

Les différentes dépendances que l'application devra extraire sont les suivantes, par ordre de priorité d'implémentation (les quatre premières sont d'importance égales et leur implémentation est de priorité très haute) :
— Dépendances par héritage/implémentation
— Dépendances par agrégation/composition
— Dépendances de signature (paramètres ou type de retour d'une méthode)
— Dépendances internes au corps d'une méthode
— Dépendances liées à la généricité (héritage d'un paramètre générique)

On considérera les classes internes comme faisant partie intégrante de leur classe englobante.

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 agrégation/composition

— **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 de signature

— **Priorité** : Forte
— **Description** : L'application devra extraire les dépendances de signature. Cette information se trouve dans la signature des méthodes. Il s'agit des paramètres ou du type de retour de celles-ci.

Extraire les dépendances internes au corps d'une méthode

— **Priorité** : Moyenne
— **Description** : L'application devra extraire les dépendances internes au corps d'une méthode. Il s'agit de l'instanciation d'un objet ou d'un appel de méthode statique.

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 mettre en place 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.

4.1.3 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 (catégorie élémentaire) à partir de la formule définie par Martin : I(**C**) = Ce(**C**) / (Ca(**C**) + Ce(**C**).

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 (catégorie élémentaire) donnée à partir de la formule définie par Martin : Dn(**C**) = |(A(**C**) + I(**C**) - 1)|.

Générer un graphe de dépendances

— **Priorité** : Forte
— **Description** : A partir des dépendances extraites de l'analyse de fichiers, l'application devra être en mesure de générer une structure de données représentative de l'interdépendance 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).
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 catégorie (package par exemple) 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.

4.1.4 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.

4.2 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.

Portabilité

— **Priorité** : Moyenne
— **Description** : L'application devra limiter son assujettissement à des outils externes, particulièrement si ceux-ci sont non portables. Des exceptions à cette règle sont envisageable si un outil s'avère pouvoir apporter un avantage important à 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.

Bibliographie

[1] R. Martin. OO Design Quality Metrics : an Analysis of Dependencies. **1994**.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, **1991**.

[3] R. Martin. *Agile Software Development - Principles, Patterns and Practices.* Pearson Education, **2003**.

[4] V. Leppanen S. Hyrynsalmi. A Validation of Martin's Metric. **2009**.

[5] V. Stanojevic D. Savic R. Lazovic S. Vlajic, M. Milic. Improving Robert C. Martin's Stability software metric. **2016**.

[6] D. Sharma G. Kaur. A study on Robert C. Martin's Metrics for Packet Categorization Using Fuzzy Logic. *International Journal of Hybrid Information Technology*, **2015**.

[7] Diomidis Spinellis. *Code Quality : The Open Source Perspective.* Addison Wesley, **2006**.

[8] R. Pressman. *Software engineering, A practitioner's approach.* Jones ans Barnett Publishers, **2000**.

[9] C. Kemerer S. Chidamber. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, **1994**.

5 Annexes

5.1 Étude préliminaire sur l'évaluation de la méthode d'analyse

TABLE 1 – Tableau comparatifs des méthodes d'analyse

Méthodes	**Code source**	**Bytecode**
Outils	Bison et Flex	ASM, Outils d'introspection du JDK
Avantages	Pas de compilation	Plus d'outils disponibles
Inconvénients	Moins d'outils existants	Nécessite de compiler

5.2 Exemple de projet annoté

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 annoté.

Listing 1 – 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 2 – Enumeration Material

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

Listing 3 – Classe Airplane

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

Listing 4 – Classe Car

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

Listing 5 – Classe Wheel

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

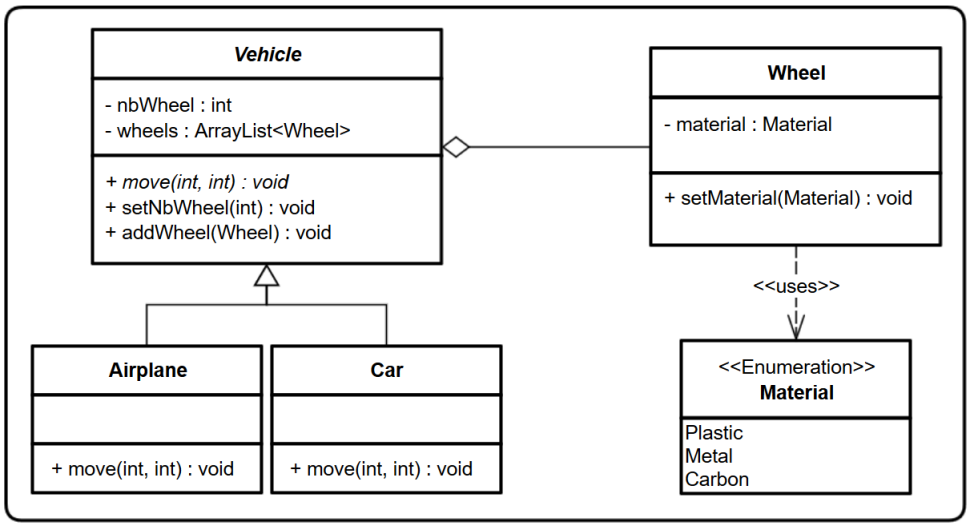


FIGURE 3 – Diagramme de classe du projet exemple

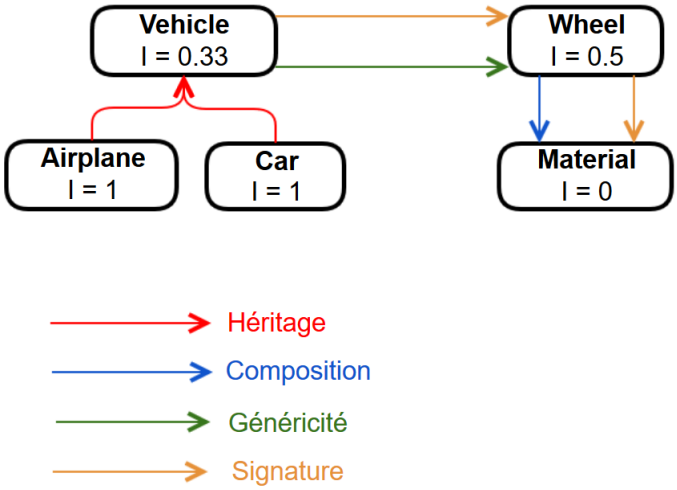
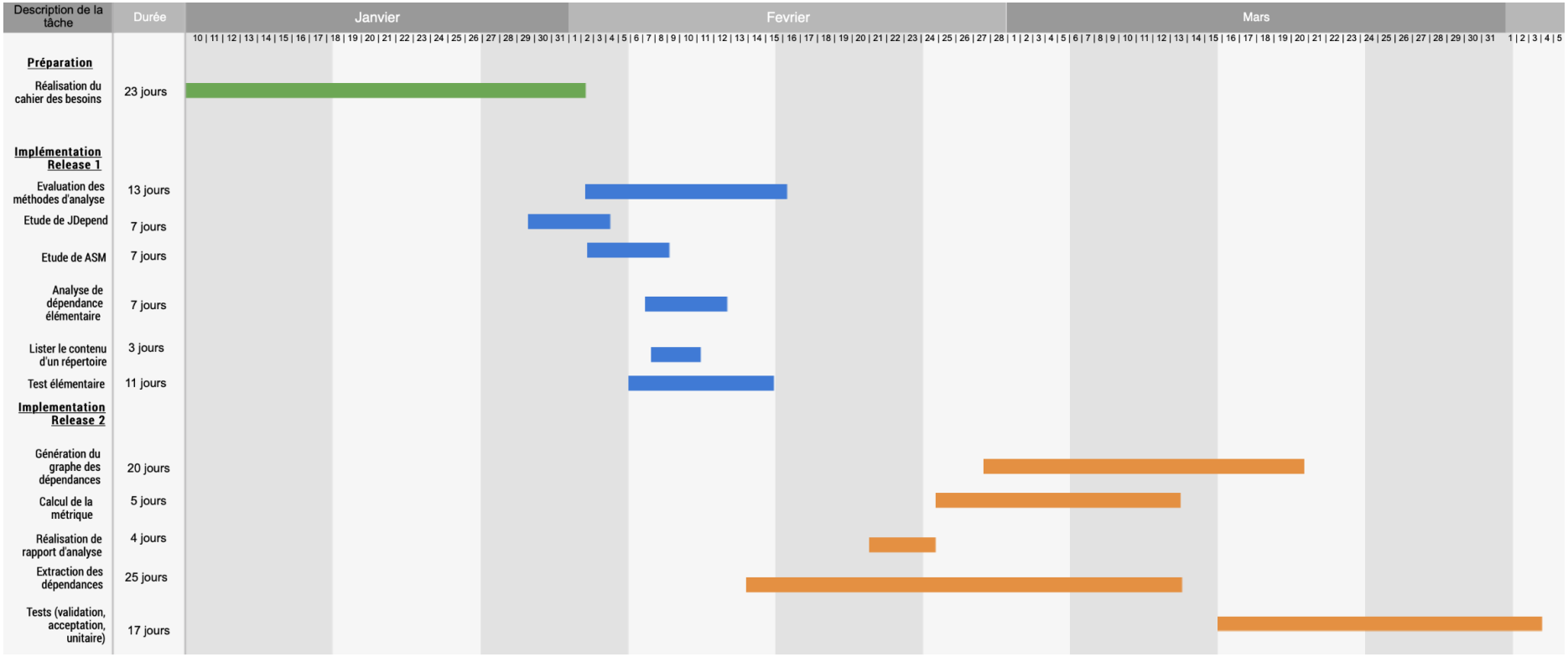


FIGURE 4 – Graphe des dépendances du projet exemple

TABLE 2 – Tableau exposant les métriques

Catégorie	Ca	Ce	A	I	Dn
Vehicle	2	1	0.33	0.33	0.33
Material	1	0	0	0	1
Airplane	0	1	0	1	0
Car	0	1	0	1	0
Wheel	1	1	0	0.5	0.5

5.3 Diagramme de Gantt


Description de la tâche
Durée
Janvier
Fevrier
Mars
10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 1 | 2 | 3 | 4 | 5
Préparation
Réalisation du cahier des besoins
23 jours
Implémentation Release 1
Evaluation des méthodes d'analyse
13 jours
Etude de JDepend
7 jours
Etude de ASM
7 jours
Analyse de dépendance élémentaire
7 jours
Lister le contenu d'un répertoire
3 jours
Test élémentaire
11 jours
Implementation Release 2
Génération du graphe des dépendances
20 jours
Calcul de la métrique
5 jours
Réalisation de rapport d'analyse
4 jours
Extraction des dépendances
25 jours
Tests (validation, acceptation, unitaire)
17 jours