UNIVERSITÉ DU QUÉBEC À MONTRÉAL

CONCEPTION ET DÉVELOPPEMENT D'UN MODULE DE CALCUL DES MÉTRIQUES POUR L'ÉVALUATION DE LA QUALITÉ DES PACKAGES DANS LES APPLICATIONS ORIENTÉES-OBJETS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

Par

Haythem Ben Ismail

Septembre 2013

À la mémoire de mon père,

à ma mère, à mon oncle,

et à toute ma famille

Résumé

Les dépendances entre les packages dans un système orienté-objets posent souvent des problèmes critiques lors d'un processus de maintenance. Le modèle conceptuel que les développeurs établissent durant la phase de conception n'est pas souvent respecté durant la phase de développement. Par conséquent, des défauts de conception sont introduits en terme de dépendances inter-packages. Ces défauts impactent gravement le développement de nouvelles fonctionalités, et empêche la maintenance et l'évolution du code source de manière fluide. L'analyse et la détection automatique de ces défauts sont donc des facteurs clefs dans l'évaluation de la qualité des systèmes orientés-objets, et facilitent considérablement la phase de maintenance et d'évolution.

Des approches et des outils d'analyse de la qualité de la modularisation ont été proposés dans la littérature. Ces approches sont basées pricipalement sur une analyse du modèle orienté-objet associé aux systèmes orientés-objets, que l'on voudrait analyser. L'évaluation de la qualité des packages dans le paradigme de l'orienté-objets se manifeste à travers les dépendances internes et externes d'un package. À cet effet, plusieurs principes de conception sont proposés pour l'organisation de ces dépendances dans le but d'avoir une bonne structuration des packages. La validation de ces modèles suivant ces principes nécessite un ensemble de métriques qui permettent d'avoir des mesures quantitatives sur la structuration des packages. Un consensus s'est forgé dans la littérature autour de quatre métriques principales pour les packages, la cohésion, le couplage, l'instabilité et l'abstraction.

Ainsi, nous proposons dans ce travail, un module de calcul de métriques pour l'évaluation des packages dans les systèmes orientés-objets, intégré au sein d'une plate-forme d'extraction et de calcul de métriques, "*Boap FrameWork*". Le calcul de métriques dans la plate-forme est basé sur un méta-modèle, établi à partir du code source des systèmes à analyser. L'implémentation des métriques est précédée par une étude approfondie de l'architecture des packages dans les modèles orientés-objets, et la définition d'un ensemble de métriques complémentaires aux métriques principales des packages.

Nous appliquons et validons nos métriques sur des systèmes libres, afin de montrer leur efficacité, suivant l'approche adoptée, dans un environnement réel.

Mots-clés : orienté-objet, package, méta-modèles, métriques, dépendance.

REMERCIEMENTS

Pas de patience, pas de science.

Jean-Pierre Jarroux

Je tiens tout d'abord à remercier chaleureusement mon directeur de recherche, Monsieur Hakim Lounis, et lui adresser ma plus profonde gratitude pour l'apprentissage et l'accompagnement durant tout ce travail de recherche. Je le remercie énormément, pour son appui moral et financier, sa disponibilité et sa gentillesse.

Je remercie également Mr El-Hachemi Alikacem, agent de recherche sénior au Centre de Recherche Informatique de Montréal, d'avoir accepter ma supervision en tant que co-directeur tout au long de ce travail. Je le remercie pour sa patience, pour son enthousiasme et pour m'avoir donner la chance de côtoyer un personnel expert au sein du CRIM.

J'aimerai remercier Mme Naouel Moha, Mr Etienne Gagnon et Mr Abdelatif Obaid, pour leur contribution indirecte dans ce travail, par leurs enseignements durant ma scolarité.

Enfin, tous mes remerciements s'adressent à ma famille, qui m'a supporté tout au long de ce projet de recherche, en particulier ma mère, ma soeur, mon frère et mon oncle.

Table des matières

Table des figures

Liste des tableaux

LISTE DES ACRONYMES

Acronyme	Signification
URI	Unifor Resource Identifier
DIP	Dependency Inversion Principle
OCP	Open Close Principle
GUI	Graphical User Interface
ISP	Inversion Stability Principle
OOD	Oriented-Object Design
RMD	Relative Module Dispersion
AMM	Average Module Membership
DIT	Depth of Inheretence Tree
NOC	Number Of Child
WMC	Weighted Method per Class
OO	Orienté Objet

Introduction

L'augmentation remarquable de la complexité des logiciels d'un point de vu exigences et développement, ainsi que l'évolution rapide des langages de programmation Orientés Objet (OO) représentent un défi majeur pour les développeurs et les mainteneurs. Or, une distribution des systèmes logiciels, en sous-systèmes permet de mieux gérer ces défis[ADSA09].

Dans le paradigme de l'orienté-objet, les sous-systèmes sont représentés par le concept des packages. En effet, les packages représentent un niveau d'abstraction plus élevé qui donne une sémantique implicite au programme via sa modularisation [SSP07]. Une structuration modulaire, permet une évolution plus fluide des systèmes OO face aux nouvelles exigences.

Les packages jouent un rôle crucial pour comprendre, faire évoluer et maintenir un système logiciel à large échelle. Les expériences ont montré que les modifications constantes des exigences et de l'environnement dégradent la qualité de la modularisation dans les systèmes OO. Par conséquent, il est primordial pour les mainteneurs d'utiliser un mécanisme qui leurs permet de comprendre la structuration des packages, et la mesure de la qualité de cette structuration. Cependant, la nature abstraite des packages représente un obstacle majeur dans la description des métriques nécessaires pour effectuer ces mesures.

Le couplage et la cohésion ont été les principales métriques pour prédire la qualité des packages durant une décennie. Mais, le développement rapide des systèmes logiciels OO et des langages OO, ont remis en question l'efficacité de ces métriques. L'ambiguité des résultats est souvent le problème majeur liée à ces métriques. D'autres métriques de packages ont été proposé dans la litérature mais elles n'ont pas été suffisement convainquantes pour qu'elle soit adoptées à large échelle. D'autre part, R Martin a proposé deux métriques complémentaires au couplage et à la cohésion, à savoir, l'instabilité et l'abstraction. Bien que la définition de ces deux métriques soit claire, leur interprétation dans un environnement réelle reste toujours ambigu selon le cas.

Un autre aspect problématique par rapport aux métriques de packages, est la difficulté de concevoir et d'implémenter ces métriques dans un langage OO, sans perdre le sens même de la mesure que nous voulons avoir. Ceci est du à l'écart entre le modèle du langage sur lequel les métriques sont définies et le modèle du langage d'implémentation.

L'utilisation des modèles dans un processus d'évaluation de la qualité de la modularisation dans les systèmes OO, s'est avéré beaucoup plus efficace que de se référer au code source directement. À cet effet, les métriques définies ne seront plus liées directement au code source mais au modèle qui représente le code source. Ceci, permet d'avoir des résultats plus précis et qui reflètent la réalité.

Les métriques proposées pour la mesure de la qualité des packages sont nombreuses. Le couplage et la cohésion sont les deux métriques proposées comme facteurs principaux pour l'évaluation de la qualité d'un package, en terme de dépendances internes et externes [AK06]. D'autre part, la stabilité et l'abstraction ont été proposées comme des métriques complémentaires, au couplage et à la cohésion. L'expérience nous a montré que l'application de ces métriques dans un environnement réel nécessite la mise en place d'un mécanisme fiable qui permet d'avoir ces mesures automatiquement. Par exemple, Goulao et al. [eAG01] proposent une plateforme qui se base sur une analyse par *Cluster* [1] pour l'amélioration du niveau de couplage et de cohésion d'un système OO. Dernièrement, les recherches se sont tournées en majorité vers l'utilisation de la notion de méta-modèles, afin d'isoler la couche de description et de calcul des métriques, de l'environnement d'exécution des systèmes logiciels en cours d'analyse [DG09, MP06, MP08].

Il est évident que durant le cycle de vie d'un logiciel, il y a une certaine nuance entre la phase de conception et la phase de développement. En effet, le schéma conceptuel conçu durant la phase de conception n'est plus le même après la phase de développement. À cet effet, il est plus approprié de se concentrer sur les modèles OO pour la réalisation d'un produit de bonne qualité en terme de modularisation.

De nos jours, les outils de calcul et d'extraction des métriques sont nombreux, mais leur fiabilité vis à vis des attentes des utilisateurs est souvent critiquée. En effet, les spécification des métriques dans ces outils manquent souvent de précision. Ceci est dû aux contraintes imposées par le langage d'implémentation de ces métriques. Dans ce contexte, Alikacem et Sahraoui [AS09] proposent un outil de calcul et d'extraction des métriques, *"Boap FrameWork"*, basé

1. *http ://fr.wikipedia.org/wiki/Clustering*

sur un langage de description des métriques nommé PatOIS, qui permet la manipulation des données extraites à partir du code source. Par la suite, ces données sont représentées dans un méta-modèle OO générique, indépendamment du langage de programmation.

Les métriques des packages dans le paradigme de l'orienté objet ont été souvent un sujet controversé par plusieurs chercheurs. En effet, à cause de la nature abstraite des notions de dépendances relatives aux packages, il est très difficile de définir des métriques qui peuvent quatifier ces notions. Dans le but de simplifier cette tâche, R.Martin a proposé un ensemble de principes qui doivent être respectés pour que la modularisation d'un système OO soit considérée comme satisfaisante [Mar03]. Dans ce cas, les métriques serviront à mesurer le niveau d'application de ces principes, dans la conception d'un modèle OO. Certes, l'ambiguité de métriques de packages reste toujours une problématique cruciale pour les maintaineurs, en particulier sous l'évolution rapide des langages OO. À cet effet, nous avons proposé dans ce travail, l'intégration d'un module de calcul de métriques pour les packages, dans la plate-forme d'extraction des métriques "*Boap FrameWork*" citée précedemment.

Dans ce mémoire, nous allons discuter la description et l'implémentation d'un ensemble de métriques qui permttent de mesurer la qualité des packages d'un système OO, d'une manière fiable et précise, indepedemment du langage de programmation. Actuellement, la plate-forme ne gére qu'un nombre limité de métriques basiques au niveau système, classes et méthodes. Par conséquent, nous allons étendre la plate-forme pour gérer les métriques des packages. Pour celà, nous devons prendre en compte trois aspects importants :

- Quelles sont les données qui doivent être extraites à partir du code source, pour quantifier les métriques de packages.
- Quels sont les choix optimaux pour la modélisation de ces données dans le méta-modèle OO générique.
- La description des métriques implémentées doit être fondée sur des bases solides et parfaitement adéquates avec les principes de conception des packages.

La contribution principale de ce travail se résume par le fait que nous avons offert par le biais de la plate-forme de description et d'extraction des métriques implémentée par Alikacem et al. [AS09], une bibliothèque de métriques pour le langage PatOIS, qui permettra aux utilisateurs de mesurer la qualité de la modularisataion d'un système OO. De plus, les choix pris durant l'implémentation pour la description de ces métriques, ont été argumentés suivant notre perception.

Dans le premier chapitre, nous présentons une introduction à la structuration des packages et leurs métriques dans le paradigme de l'orienté-objet. On s'intéresse particulièrement aux différentes configurations possibles des packages dans un système OO. Par la suite, on présente les différents principes de conception reliés aux dépendances internes et externes des packages. Finalement, on discute les métriques principales des packages, à savoir, la cohésion, le couplage, la stabilité et l'abstraction.

Le deuxième chapitre traite de plusieurs travaux connexes à notre sujet de recherche. En premier lieu, on discute quelques travaux qui se basent sur les méta-modèles pour la représentation des informations relatives à la structuration des packages. Ensuite, on présente les principes de conception des packages, proposées pour valider la modularisation dans les systèmes OO. Finalement, on discute les métriques de packages définies dans la littérature. Celles-ci sont divisées on deux catégories, les métriques statiques et les métriques dynamiques.

Le troisième chapitre est consacré à l'utilisation des méta-modèles pour la définition des métriques. Tout d'abord, on explique la relation entre un méta-modèle et les métriques. Ensuite, on présente le processus de *"mapping"* au sein de la plate-forme, et le méta-modèle adopté pour le calcul d'un ensemble de métriques génériques niveau programme, classes et méthodes. Finalement, on discute les extensions apportées au méta-modèle en question. Ces extensions concernent quelques aspects d'un programmes OO, et les dépendances inter-modulaires qui peuvent exister dans un système OO.

Le quatrième chapitre décrit les étapes de l'intégration du module de calcul des métriques de packages au sein de la plate-forme. Premièrement, on présente l'outil de description et d'extraction des métriques *"Boap FrameWork"*. Deuxiemement, on décrit les différents mécanismes, y compris la syntaxe du langage de description des métriques, PatOIS. Par la suite, on présente le module évaluateur qui joue le rôle de l'interpréteur du langage. Il exécute les requêtes de calcul des métriques sur l'instance du méta-modèle générée durant la phase de *mapping* du programme OO à analyser. Finalement, on défini les métriques implémentées pour la mesure de la qualité des packages. Celles-ci sont répertoriées en trois types, les métriques primitives, complexes et les métriques basées sur d'autres métriques.

Le cinquième chapitre contient des études de cas sur quelques programmes OO implémentées avec le langage *Java* : *CPL*, *PADL* et *PayrusForTest*. Tout d'abord, on teste la validité de l'instance du méta-modèle générée avec le code source. Ensuite, on discute les résultats des différents types de métriques implémentées, par rapport aux trois cas de test. Finalement, on

présente le graphe de dépendances niveau packages et classes, généré suivant le langage DOT, pour le cas *PayrusForTest*.

Nous utilisons les termes *modules* et *packages* dans la suite du mémoire, pour designer un sous-système d'une manière abstraite, indépendamment du sens technique des deux mots.

Chapitre 1

Analyse de la structure des packages et leurs relations

1.1 Introduction

Les packages jouent un rôle majeur dans les systèmes orientés objet. En effet, ce composant est un conteneur de classes qui représente une logique sémantique reflétant le schéma conceptuel élaboré pendant la phase de conception.

L'augmentation de la complexité des systèmes OO, et le changement remarquable des besoins fonctionnels, proportionnel aux optimisations apportées sur les langages OO, a poussé les chercheurs à s'investir d'avantage dans le domaine de l'évaluation de la qualité des modèles OO.

Sur un premier plan, l'évaluation d'un modèle OO se manifeste à travers la qualité de sa modularisation. À cet effet, les chercheurs commencent a s'intéresser de plus près aux packages, qui sont la représentation concrète d'un module en termes de structuration et relations. En effet, les packages sont de plus en plus importants vu qu'ils jouent un rôle majeur d'un point de vue syntaxique et sémantique.

L'analyse de la structure des packages et de leurs relations devient de plus en plus le sujet de plusieurs recherches, qui essayent d'oeuvrer pour la conception d'un modèle OO, de bonne qualité. En revanche, la nature abstraite des packages rend le processus d'établir un consensus clair très critique. Ainsi, R.Martin [Mar03] a réussi à établir des principes de base que l'on peut référer, pour la conception des modèles OO.

Les définitions des métriques qui visent à appliquer les différents principes de la conception des packages sont presque approuvées par la majorité des chercheurs. Ces métriques se manifestent par la cohésion, le couplage, l'instabilité et l'abstraction. Or, cela s'est avéré insuffisant, vu que les mesures retournées par ces métriques sont imprécises. Par conséquent, les interprétations de ces métriques varient énormément d'un travail à un autre.

Afin de rendre les notions autour de ces composants le plus claires possible, nous présentons dans ce chapitre une analyse structurelle et relationnelle des packages, ainsi que les métriques définies pour ce type de composant abstrait.

Nous allons présenter dans la première partie, le rôle des packages d'un point de vue conceptuel et sémantique. Par la suite, nous présentons les principes relatifs aux relations des packages sur le plan conceptuel. Et finalement, nous définissons les métriques que nous avons adoptées pour la mesure de la qualité des packages.

1.2 Architecture des packages

1.2.1 Définition

Les packages sont des entités complexes qui jouent plusieurs rôles dans un système OO. En effet, ils représentent une couche abstraite de haut niveau au dessus des classes, avec des caractéristiques particulières. Dans un modèle OO, un package représente un sous-système qui définit une tâche bien déterminée de l'ensemble des tâches à exécuter par le système en cours d'analyse.

D'un point de vue technique, les packages sont des répertoires créés instantanément lors de la compilation au niveau du disque dur. Ces répertoires contiennent les fichiers des classes qu'ils incluent. D'ailleurs, on les considère généralement comme étant des composants statiques par rapport aux autres composants, comme les objets dans le paradigme OO. D'où, l'explication du surnom que l'on donne aux packages dans le jargon informatique, les boites noires (BlackBox).

1.2.2 Le rôle des package dans les systèmes OO

Les différents rôles qu'un package joue dans un système, font de lui une entité complexe très difficile à comprendre. En effet, les packages permettent aux mainteneurs et aux développeurs de comprendre plus facilement les différentes parties fonctionnelles d'un système OO. Généralement, les packages jouent un rôle sémantique très important dans les systèmes OO. Ils regroupent un ensemble d'objets qui ont la même sémantique d'un point de vue fonctionnel.

D'un point de vue conceptuel, les packages protègent les objets des accès non sécurisés de la part des autres objets qui ne partagent pas forcement la même sémantique en termes de fonctionnalités. D'un point de vue structurel, les packages offrent une meilleure structuration du code dans les programmes OO. En effet, les packages assurent une meilleure visibilité pour les développeurs. Ils l'aident à mieux identifier les emplacements des entités, durant la phase de développement et celle de maintenance.

Un package peut être soit producteur soit client (consommateur) ou les deux en même temps. En effet, nous présentons dans ce qui suit les principaux rôles qu'un package peut jouer, illustrés par la figure ci-dessous. La *figure1.2.1* représente un exemple typique d'un patron de conception très populaire, le MVC (Modèle-Vue-Contrôleur).

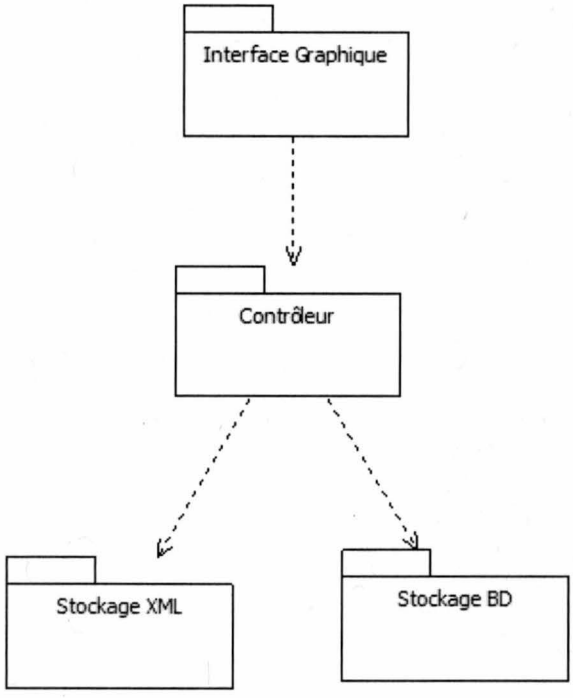


FIGURE 1.2.1 – Illustration des différents rôles des packages suivant le patron de conception MVC

Le package *interface graphique* : représente l'ensemble des classes qui constituent les différents composants statiques d'une interface graphique quelconque.

Le package *contrôleur* : contient l'ensemble des classes qui contrôlent les différents composants de l'interface graphique (bouton, items, menus, etc.).

Les packages *BD, XML* : contiennent les classes qui modélisent l'ensemble des données à manipuler.

Package producteur

Un package producteur est un package qui propose un ensemble de services à un ou plusieurs autres packages dans un système OO. D'après *la figure1.2.1*, le package *interface graphique*, est un package producteur pour le package *contrôleur*. En effet, ce package met à la disposition des classes du package *contrôleur*, l'ensemble des composants graphiques, dont ce dernier aura besoin pour associer les actions nécessaires, relatives à chaque composant graphique. Par

conséquent, le package *interface graphique* joue le rôle d'un producteur de service pour le package *contrôleur*.

Package Client

Un package client est un package qui utilise les services d'un ou plusieurs packages. Dans notre exemple, les packages *XML* et *BD* sont des packages clients pour le package *contrôleur*. En effet, ils se servent des résultats retournés par ce dernier et les modélisent sous forme de données structurées (package XML), ou sous forme de données brutes (package BD).

Package Client / Producteur

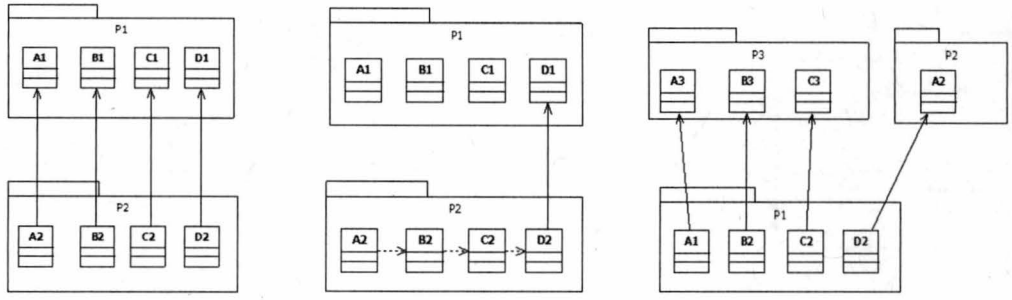
D'autre part, les packages peuvent jouer à la fois le rôle d'un package producteur et le rôle d'un package client. Dans la figure ci-dessus, le package *contrôleur* joue le rôle d'un package producteur pour les packages *XML, BD* et client par rapport au package *interface graphique*.

1.3 Configuration des packages

Les packages ont un rôle organisationnel très important dans les systèmes OO. Ils reflètent une organisation bien précise de l'ensemble des classes, parmi plusieurs organisations possibles. En effet, chaque organisation ou structuration des packages représente une définition sémantique unique de l'ensemble des entités du système.

D'ailleurs, comme nous l'avons précisé dans la section précédente, un package est une structure conceptuelle et architecturale de haut niveau. Elle joue le rôle d'une boite statique qui encapsule un morceau quelconque de code source. Donc, Il est tout à fait légitime qu'on trouve des connexions entres les packages, où les entités communiquent entre elles de plusieurs manières. La *figure1.3.1*, est un schéma qui représente plusieurs configurations possibles des packages à travers le même ensemble de classes.

Nous présentons dans la *figure1.3.1* trois configurations différentes des packages. En effet, dans le modèle ci-dessous nous avons choisit de garder le même nombre d'entités, dans le but de démontrer que la sémantique d'un modèle OO ainsi que son organisation via le nombre de packages, supporte plusieurs configurations possibles.



(a) (b) (c)

FIGURE 1.3.1 – Différentes configurations de packages inspiré par[Abd09]

La *figure1.3.1(a)*, montre que toutes les classes du package $P2$ référencent les classes du package $P1$. Dans la *figure1.3.1(b)*, le package $P2$ contient des classes qui sont référencés en interne. Ceci, est très important à comprendre dans un processus d'évaluation de la qualité d'un modèle OO. En effet, ça permet aux mainteneurs d'anticiper l'impact d'une modification quelconque sur l'ensemble des classes du package $P1$.

Dans la *figure1.3.1(c)*, les classes du package $P1$ référencent directement les classes du package $P3$ et $P2$. Selon R.Martin, le package $P1$ de la *figure1.3.1(c)* nécessite une importation des deux packages $P2$ et $P3$. Or, dans la *figure1.3.1(a),* le package $P2$ n'importe que le package $P1$. Ceci implique que l'effort de maintenance du modèle de la *figure1.3.1(c)* est plus important que celui de la *figure1.3.1(a)*.

Pour résumer, la modularisation dans un modèle OO supporte plusieurs configurations possibles. Par conséquent, une question s'impose : laquelle des configurations est la plus adéquate en terme de qualité de modularisation ?

1.4 Les dépendances

L'évaluation de la qualité d'un modèle OO se base sur les interdépendances entre les sous-systèmes du modèle. En effet, un modèle qui supporte la communication externe et interne entre ses sous-systèmes, tels que les modèles OO, a souvent des problèmes de rigidité, à cause de ces communications.

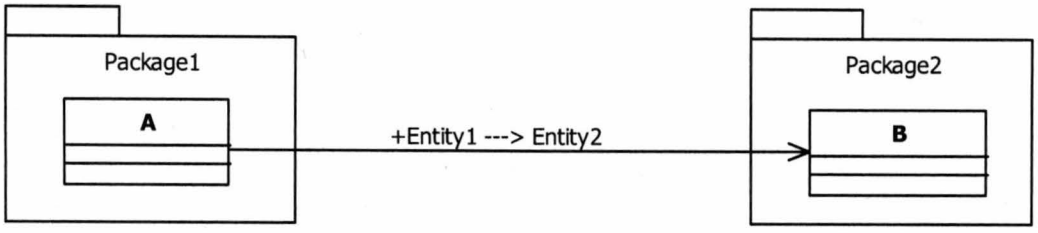


FIGURE 1.4.1 – Dépendance niveau entité

Dans les normes internationales, un modèle fragile perd sa crédibilité en terme de maintenance, ce qui affecte la qualité du modèle en générale. Afin de mieux étudier la qualité des modèles OO, nous devons analyser les interdépendances qui peuvent exister entre les sous-systèmes de ce modèle.

Dans un modèle OO, les sous-systèmes sont représentés par les modules, plus précisément, les packages. Tel que nous l'avons précisé dans la section précédente, les packages sont de simples conteneurs abstraits qui jouent essentiellement un rôle sémantique et organisationnel par rapport aux entités qu'ils encapsulent.

Les entités sont les composants concrets des sous-systèmes. Par conséquent, toute dépendance d'un sous-système à un autre, passe essentiellement par les entités. À cet effet, et dans le but de bien comprendre la notion des interdépendances entre les sous-systèmes, nous devons descendre d'un niveau d'abstraction et étudier d'abord les dépendances entre entités.

1.4.1 Les dépendances niveau entités

Dans un système OO, une relation de dépendance existe entre deux entités, si et seulement si, une entité envoie un message à une autre entité. L'envoi d'un message d'une entité à une autre est considéré comme une demande d'accès vers une information contenue dans l'entité ciblée par la dépendance.

Afin de mieux comprendre la notion de la dépendance entre deux entités, nous présentons dans la *figure1.4.1*, la relation de dépendance entre deux entités, telle que nous la percevons.

D'après la figure, nous avons une relation de dépendance entres deux entités A et B notée (A→B). L'entité A envoi un message à l'entité B. Autrement dit, A essaye d'accéder à une

information dans B. Ainsi, A dépend de B, car toute modification au niveau de B aura des répercussions sur A. Nous appelons A l'entité origine et B l'entité destination. D'un point de vue analytique, cette dépendance est interprétée suivant deux vues. Pour l'entité A, cette dépendance est une dépendance sortante vers B. Or, pour l'entité B c'est une dépendance entrante de A. Ceci est très important en terme de relation dans un modèle OO, car lorsque nous avons une relation de dépendance sortante, (A→B), ceci implique que A est instable par rapport à B. Dans le sens inverse, B est stable par rapport à A.

Tel que nous l'avons montré précédemment, un ensemble d'entités peut avoir plusieurs configurations par rapport aux packages qui les encapsulent dans un modèle OO. Ainsi, une relation de dépendance peut être interprétée de plusieurs façons différentes, dépendamment du contexte modulaire où les deux acteurs de la dépendance sont hébergés.

1.4.2 Les dépendances au niveau des packages

Dans la partie suivante, nous allons étudier les relations de dépendance d'un niveau d'abstraction supérieur à celui des entités. La dépendance entre deux packages passe par les entités car réellement, il n'existe pas de dépendance directe entre deux packages, vu que c'est un composant abstrait dans le modèle OO.

La *figure1.4.2* représente une vue générique de l'ensemble des dépendances qui peuvent exister dans un modèle OO. Le package P1 est le package visé par l'analyse. Il contient un ensemble de 6 entités où chaque entité est reliée par une ou plusieurs relations de dépendance avec le reste des entités du système.

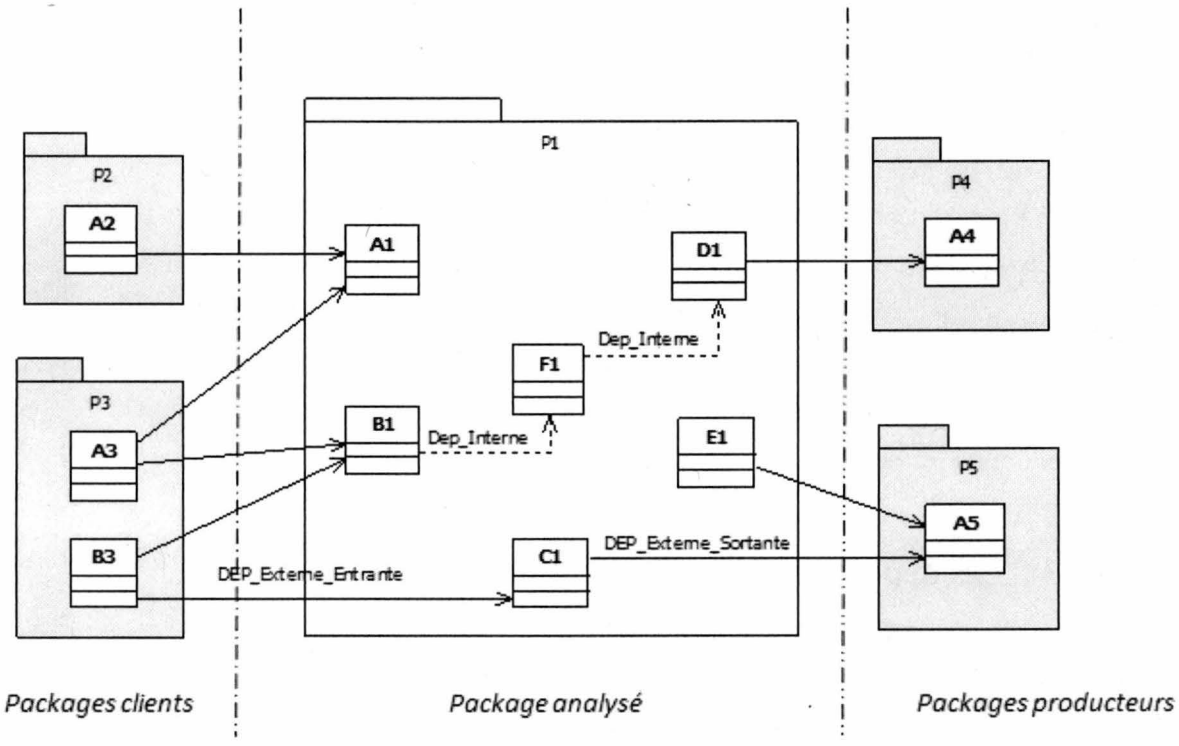


FIGURE 1.4.2 – Dépendances inter et intra packages

Si une relation de dépendance existe entre deux entités par exemple ($A3 \rightarrow A1$), cela implique qu'une relation de dépendance entre leurs packages respectifs existe aussi ($P3 \rightarrow P1$). Cette relation est considérée comme une dépendance externe pour le package $P1$. Si une relation de dépendance implique deux entités appartenant au même package, ($B1 \rightarrow F1$), la relation est considérée comme une dépendance interne pour le package $P1$.

On défini les deux types de dépendances comme suit :

Les dépendances externes : c'est une dépendance entre deux entités appartenant à deux packages différents. Par exemple, ($B3 \rightarrow B1$) ou ($D1 \rightarrow A4$). par conséquent, ($P3 \rightarrow P1$) et ($P1 \rightarrow P4$). De plus, une relation de dépendance externe est lu dans les deux sens de la dépendance, soit au niveau des packages ou au niveau des entités. Par rapport à la relation ($B3 \rightarrow B1$), la dépendance est considérée comme une relation de dépendance *sortante* pour l'entité $B3$ et

entrante pour l'entité $B1$. Cette interprétation est la même pour leurs packages respectifs. En effet, les dépendances sortantes et entrantes ont un impact important dans l'interprétation des relations des packages dans un système OO. D'après la *figure1.4.2*, on peut conclure grâce à la notion de dépendances entrantes et sortantes que la relation de dépendance ($P3 \rightarrow P1$), implique que le package $P3$ est un package client pour le package $P1$, et inversement, $P1$ est un package producteur pour $P3$.

Les dépendances internes : c'est la dépendance entre deux entités appartenant au même package. Par exemple, ($B1 \rightarrow F1$) ou ($F1 \rightarrow D1$) sont considérées comme étant des dépendances internes pour le package $P1$. Même au niveau interne d'un package, les relations de dépendance se manifestent aussi à travers les deux sous-types, *entrantes* et *sortantes* mais cela n'est considéré qu'au niveau des entités. L'interprétation de dépendances internes en dépendances entrantes et sortantes est importante dans le cas où nous voulons analyser la structuation d'un package en interne. De plus, cette interprétation est essentielle pour la définition des métriques de dépendances au niveau interne des packages.

Pour résumer, il est très important d'identifier ces deux sous-types de dépendance dans un processus d'analyse de dépendance d'un modèle OO, car comme au niveau des entités, il existe toujours un package origine et un package destination. Par exemple, le package $P1$ est considéré comme étant un package *destination* à travers la dépendance ($B3 \rightarrow C1$). D'autre part, $P1$ est un package *origine* pour la dépendance ($C1 \rightarrow A5$).

Les packages d'interfaces : on appelle les entités qui ont des dépendances avec des entités externes au package auquel elles appartiennent, les entités d'interfaces. D'après la *figure1.4.2*, nous identifions les entités ($A1, B1, C1, D1, E1$) comme étant les entités d'interfaces du package $P1$. En effet, les entités d'interfaces jouent un rôle très important dans un package vue qu'elles assurent la communication du package avec le reste des packages dans un modèle OO. Également, et dans un processus de maintenance du modèle, ces entités sont les plus susceptibles de subir des modifications, vu qu'elles ont des dépendances avec le reste des entités du système.

1.5 Les principes de conception des packages

La rigidité des modèles OO était toujours un important challenge que les concepteurs des applications orientées objets ont toujours considéré. En effet, un modèle OO rigide implique que

n'importe quel changement au niveau d'un sous-système de ce modèle, entraine des modifications majeures dans le reste des sous-systèmes. La fragilité du modèle, dégrade remarquablement sa crédibilité ainsi que le processus de maintenance[Mar94].

Un modèle OO de mauvaise qualité ne pourra pas engendrer des modules assez robustes et indépendants pour qu'ils soient souvent réutilisés dans d'autres modèles. La dégradation de la qualité des modèles OO durant la phase de développement, est dûe essentiellement à l'absence d'un mécanisme de contrôle ou d'un standard clair. Néanmoins, prévoir des mesures fiables et efficaces qui permettent d'alerter les concepteurs sur ce genre de défaut à une étape assez anticipée est une solution fiable.

Les interdépendances entre les modules sont les principales causes de la fragilité d'un modèle. En effet, un simple changement dans un module pourra entrainer une série de modifications complexes sur le reste des modules, à condition qu'il y ait une forte dépendance avec le module en question.

Dans ce contexte, beaucoup d'études et d'analyses ont porté sur les relations entre les packages dans un modèle OO. À cet effet, nous présentons dans ce qui suit, une analyse détaillée des principes de conception des packages qui permettent aux concepteurs de bien répartir les classes à l'intérieur des packages. Le but est d'assurer la bonne qualité d'un modèle OO afin qu'li réponde aux attentes des mainteneurs en termes de précision et d'efficacité.

1.5.1 Les principes de cohésion

1.5.1.1 Release Reuse Equivalence Principle : REP

La granule de réutilisation est la granule de libération [1].

Depuis que la version de UML2.4.1 a introduit la notion des attributs URI [2] pour les packages, ils deviennent des unités de libération [3]. Par conséquent, on considère systématiquement que ces composants sont des unités de réutilisation aussi. Par définition, le niveau de la réutilisation d'un élément dépend de ces limites de libération [Mar03].

1. ***The granule of reuse is the granule of release.***
2. http ://www.uml-diagrams.org/package-diagrams.html
3. http ://docs.cray.com/books/004-3689-001/html-004-3689-001/zmpt06relpkgd4bcgpsa.html

Généralement, dans les applications OO, il existe toujours des systèmes de libération pour les packages afin d'assurer leurs réutilisation en toute sécurité. En effet, un package peut être réutilisé facilement par plusieurs utilisateurs, s'il existe une garantie de la part du développeur du package que la nouvelle version de l'élément, n'entrainera pas de répercussions sur l'élément qui sera réutilisé.

Cependant, il est tout à fait possible de considérer les classes comme unités de réutilisation et de libération. Or, ceci est très difficile à concrétiser dans la pratique, car le système de libération qui sera mis en place sera obligé de gérer un grand nombre d'unités à une échelle très réduite. Ceci pourra entrainer des problèmes au niveau des fonctionnalités de l'application.

Pour conclure, ce principe proposé par R. Martin, consiste à dire qu'une bonne structuration d'un package passe par le regroupement des classes qui sont réutilisables.

1.5.1.2 Common Closure Principle : CCP

Les classes qui changent ensemble, vont ensemble [4].

Généralement, les grandes applications orientées objets sont subdivisées en sous-tâches, réparties sur un ensemble de packages reliés entre eux. Or, on sait que plus un package contient de relations avec les autres packages, plus il est difficile à comprendre par les mainteneurs[Mar03].

Le principe CCP vise principalement à minimiser l'ensemble des packages qui nécessitent des modifications pendant le processus de libération. En effet, ce genre des packages contient des classes qui sont en relations avec d'autres classes réparties dans d'autres packages du système. D'où, la recommandation de restructurer les classes d'une façon plus adéquate.

Par déduction, nous pouvons dire qu'un modèle OO facile à maintenir, est celui qui contient le minimum possible de ce genre de packages. De plus, même si l'existence de ce genre de packages est inévitable dans les applications OO de grande envergure, il est recommandé de les répartir d'une façon plus intelligente, dans le but de minimiser l'effort lors d'un processus de maintenance.

À cet effet, le principe CCP vise ces classes et adopte une définition simple, qui consiste à regrouper les classes qui changent ensemble dans le même package. Ceci, permettra de ré-

4. ***Classes that change together, belong together***

duire l'impact des modifications apportées, sur l'ensemble du système, pendant le processus de libération de ces packages.

Par exemple, supposons qu'on ait une classe A contenu dans un package$P1$, en relation avec une autre classe B, contenu dans un package $P2$. Il est clair que toute modification sur la classe A entrainera des modifications sur la classe B. Ceci, aurait certainement moins d'impact en terme d'effort et d'organisation si les deux classes étaient dans le même package. D'où, le principe CCP, qui recommande de concevoir ces deux classes dans un seul package.

Certes, il existe toujours un certain nombre de packages qui nécessitent des modifications dans un système OO. Mais, plus le nombre de ces packages est élevé, plus la qualité du modèle OO du système est dégradée. Par conséquent, l'application du principe CCP permet de bien répartir ce genre des classes dans les packages d'un système OO.

1.5.1.3 Common Reuse Principle : CRP

Les classes qui sont réutilisées ensemble, doivent être regroupées ensemble [5].

Comme nous l'avons expliqué dans la section1.4, une dépendance qui cible un package implique que c'est une dépendance à travers une ou plusieurs classes de ce package. Par conséquent, un package qui respecte le principe CRP, implique que n'importe quelle dépendance à travers ce package est une dépendance à travers toutes les classes de ce package. En effet, le principe CRP consiste à regrouper les classes qui sont réutilisées ensemble dans le même package.

Généralement, il est très rare de trouver dans les systèmes OO des classes qui sont réutilisable individuellement. En effet, il existe toujours d'une manière ou d'une autre, une liaison avec d'autres classes du système. Donc, il est plus raisonnable de regrouper ces classes ensemble dans le même package. Ceci permettra de réduire l'effort des mainteneurs pour une éventuelle modification pendant le processus de maintenance.

Dans la pratique, les packages qui sont réutilisables (librairies, API, ...), subissent toujours des améliorations constantes. Ceci peut être nuisible en cas d'absence de l'application du principe CRP sur ces packages. En effet, si les classes réutilisables ensemble ne sont pas regroupées dans le même package, alors la réutilisation de ce package risque d'être très pénible en terme

5. ***Classes that aren't reused together should not be grouped together***

d'effort. Premièrement, chaque modification sur la version originale du package nécessitera une mise à niveau de la part de l'utilisateur vis à vis de l'ancienne version. De plus, une validation systématique s'impose, pour être sur que les modifications apportées sur le package n'ont pas de répercussions sur le reste du système.

1.5.2 Les principes de couplage

Depuis que le consensus de regrouper les classes qui changent ensemble existe, l'effort de la maintenance est réduit. En effet, lors d'un processus de libération, les développeurs se retrouvent avec un nombre de packages à modifier très réduit par rapport au nombre qui pouvait exister en cas d'absence du consensus.

Dans un processus de libération, un package en relation avec d'autres packages, nécessite des tests de validation avec le reste des packages du système. Ainsi, la compilation et la reconstruction du package au sein de son nouvel environnement s'impose.

Certes, lors du développement d'un système OO, les développeurs ont tendance à oublier les contraintes et les limites qu'ils se sont fixés lors de l'étape de conception du projet. Parfois, des liens entre les packages se créent sans aucune vérification. Ceci risque de dégrader remarquablement la qualité du modèle OO final.

Dans ce qui suit, nous allons étudier les principes de conception des packages en terme de couplage. Cette étude est très importante dans la mesure où elle nous permet de comprendre la structure des packages, ainsi que leurs rôles dans le système OO.

1.5.2.1 Acyclic Dependencies Principle : ADP

Les dépendances entre les packages ne doivent pas former des cycles [6].

Les études qui ont porté sur les dépendances inter-packages sont nombreuses. En effet, ces dépendances, peut importe leur nature, ont un impact majeur en terme d'effort lors d'un processus de libération. Depuis que les packages sont l'unité de libération, les dépendances au niveau de ces composants représentent un inconvénient majeur pour les mainteneurs.

6. ***The dependencies betwen packages must not form cycles.***

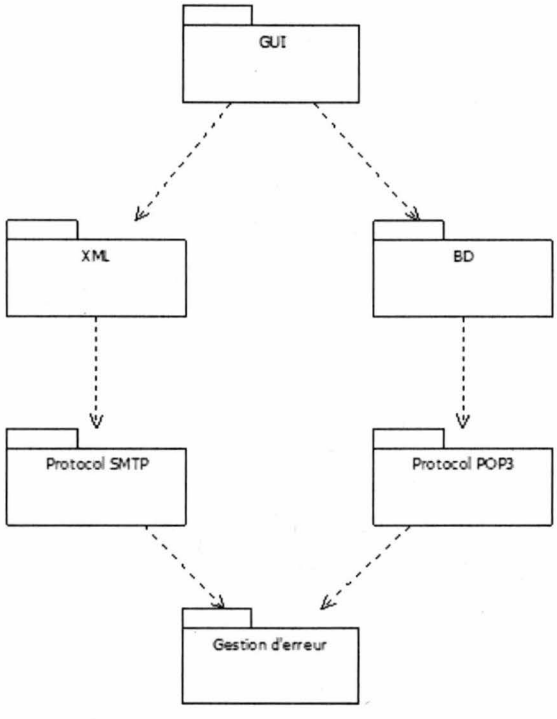


FIGURE 1.5.1 – Modèle OO d'une application de gestion de messagerie

La *figure1.5.1*, représente un modèle d'une application OO d'un module de gestion de messagerie. Il est clair que la structure du modèle est très mauvaise d'un point de vue conceptuel. En effet, elle ne respecte ni le principe DIP [7], ni le principe OCP [8]. Cependant, ce modèle reste parfait pour définir le principe ADP.

Tout d'abord, nous avons le package "GUI" qui représente l'ensemble des entités qui assurent l'affichage des informations à l'écran. Visiblement, ce package dépend directement des packages XML et BD qui représentent deux types de stockages différents. De même, ces deux packages dépendent respectivement des deux autres packages, "POP3" et "SMTP", qui représentent deux protocoles de transmission de messages. Le premier est pour la réception des messages, et l'autre, pour l'envoi.

Supposons que nous voulons libérer le package "protocole SMTP" dans la *figure1.5.1*. Ce package a une dépendance pointée vers le package "gestion d'erreur". À cet effet, le processus de libération de ce package nécessite une construction complète du package avec la dernière version du package "gestion d'erreur", ainsi que les tests de validation.

Supposons que nous voulons appliquer quelques changements sur le package "gestion d'erreur". D'après la version actuelle du modèle, ce package ne fait que gérer les erreurs relatives aux deux protocoles et il ne dépend d'aucun autre package. Donc, les modifications peuvent s'appliquer sur ce package sans avoir de répercussions sur le reste des packages du système.

En revanche, si on décide d'afficher les messages d'erreurs relatifs aux deux protocoles sur l'écran, nous devons modifier le modèle. En effet, un objet du package "GUI" recevra le message d'erreur à afficher à partir du package "gestion d'erreur" et l'affiche instantanément. Du coup, le développeur sera obligé d'établir une nouvelle dépendance entre le package de "gestion d'erreur" et celui de l'interface homme-machine "GUI". La *figure1.5.2* représente le nouveau modèle établi.

L'ajout de la nouvelle dépendance peut s'avérer catastrophique en cas d'une éventuelle libération du package "protocole SMTP". En effet, toute modification dans ce package, nécessitera la reconstruction des packages suivants : GUI, XML, BD, SMTP, POP3 et gestion d'erreur, ainsi que leurs validation via des tests. Par conséquen, ceci risque d'alourdir la tâche des mainteneurs.

7. Dependancy Inversion Principle : http ://en.wikipedia.org/wiki/Dependency_inversion_principle
8. Open Close Principle : http ://en.wikipedia.org/wiki/Open/closed_principle

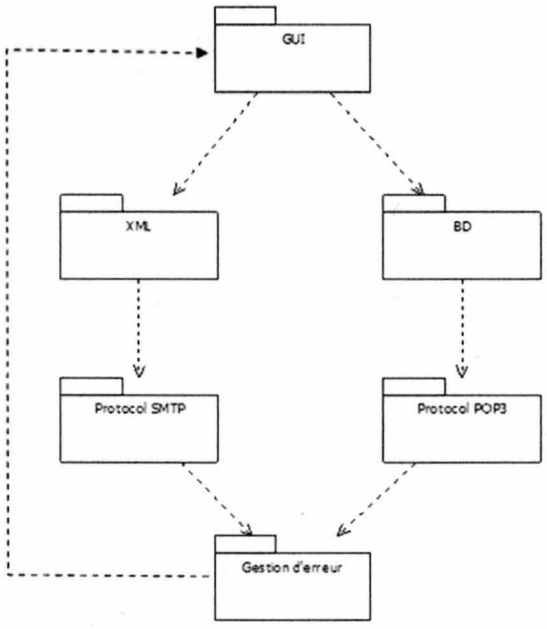


FIGURE 1.5.2 – Création d'un cycle de dépendance dans le modèle

La dépendance que nous avons ajoutée du package " Gestion d'erreur " vers le package " GUI ", a créé un cycle de dépendance entre ces deux packages. Le fait est que lors d'un processus de libération de n'importe quel package, la mise à niveau et la validation s'applique sur tous les packages qui forment le cycle de dépendance. Dans notre exemple de la *figure1.5.2*, nous remarquons qu'un simple lien de dépendance entre deux packages, a crée tout un cycle qui contient tout l'ensemble des packages du modèle. Ceci est vraiment inacceptable en terme de coût de maintenance.

Dans la pratique, les dépendances cycliques sont souvent présentes dans les modèles OO. En effet, il existe deux solutions possibles pour briser ce genre de dépendances :

–» Ajouter un nouveau package entre les deux packages formant le cycle de dépendance.

–» Utiliser la notion de DIP et ISP[Mar94], pour briser les cycles de dépendance.

La *figure1.5.3* illustre la création d'un nouveau package " Message Display " qui joue le rôle de l'intermédiaire entre les packages " GUI " et " Gestion d'erreur ". Les dépendances sont redirigées vers le nouveau package tel que c'est illustré dans la figure ci-dessous.

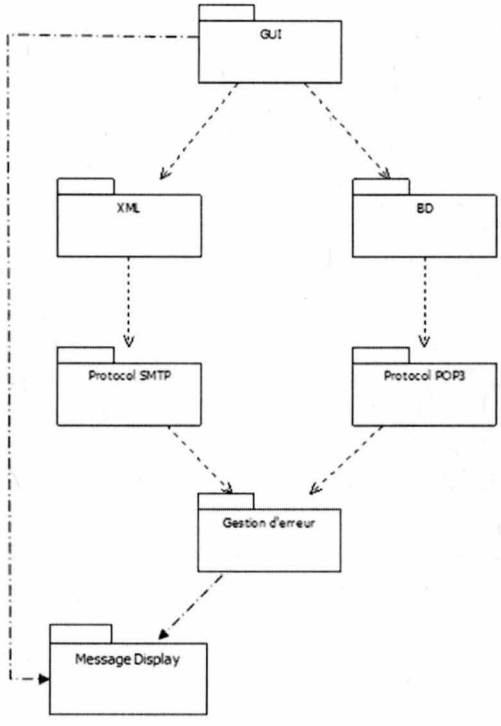


FIGURE 1.5.3 – Briser le cycle avec un package intermédiaire

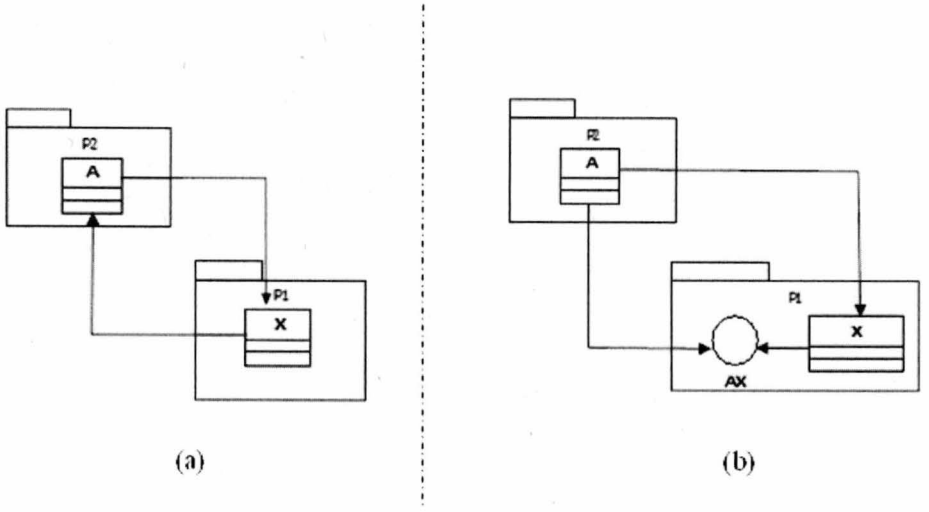


FIGURE 1.5.4 – Briser le cycle de dépendance avec le principe DIP

L'idée générale de la solution, c'est d'ajouter un autre niveau d'abstraction vers le bas. En effet, on sélectionne les objets qui communiquent entre eux, appartenant aux packages " GUI " et " Gestion d'erreur " et on les replace dans le nouveau package. D'un point de vue organisationnel, cette solution est très bénéfique vue qu'elle permet de mieux structurer le modèle. De plus, elle améliore remarquablement la qualité du modèle en respectant le principe de cohésion CCP. Cependant, dans le cas d'un système très complexe avec des centaines de packages et des milliers de classes, cette solution risque d'être coûteuse en terme de main d'œuvre.

La *figure1.5.4* présente la deuxième solution pour briser les cycles de dépendance entre les packages. En effet, l'idée est d'inverser l'une des deux dépendances suivant la notion DIP définie par R.Martin. Dans la *figure1.5.4(a),* on a deux packages $P1 et P2$ qui forment une dépendance cyclique directe via les deux classes A et X, $(A \rightarrow X)$ et $(X \rightarrow A)$. Pour briser le cycle, on introduit tel que cela apparait dans la *figure1.5.4 (b)*, une interface AX, qui contiendra toutes les méthodes dont X a besoin, ainsi, $(X \rightarrow AX)$. D'autre part, A implémente toutes les méthodes dont elle a besoin, et qui sont définies dans l'interface AX. Ainsi, la dépendance est inversée de A vers AX.

En effet, le choix de placer l'interface AX dans le package $P1$ et non pas dans $P2$ est justifié par le fait que l'interface est souvent conçu dans le package où elle sera utilisée et non pas là où elle sera implémentée. C'est un patron de conception utilisé dans beaucoup d'applications OO.

1.5.2.2 Stable Dependencies Principle : SDP

Une dépendance pointe vers la direction de la stabilité [9].

Souvent dans les modèles OO, les packages ont des dépendances avec les autres packages. Comme nous l'avons déjà précisé dans la section1.4, les dépendances peuvent être soit entrantes soit sortantes. Un package est très difficile à changer lorsque il y a plusieurs autres packages qui dépendent de lui. On l'appelle un package très responsable[Mar94].

Plus le nombre de dépendances entrantes d'un package est grand, par rapport à ses dépendances totales, plus il est stable. En effet, n'importe quel changement au niveau de ce package provoquera un impact sur l'ensemble des packages qui dépendent de lui. D'où, la notion du principe SDP que nous présentons. Ce principe consiste à respecter le fait que lors de la création d'une relation de dépendance entre deux packages, il faut que la dépendance soit dirigé vers un package plus stable que celui d'origine. Autrement dit, il ne faut pas que le package de destination soit impliqué dans des dépendances externes, plus nombreuses que celles de l'origine.

La *figure1.5.5(a)* met en évidence l'application du principe SDP. On remarque que les packages$X$ et Y dépendent du package A. Or, A n'a aucune autre dépendance. Par conséquent, on peut déduire que le package$A$ est plus stable que les deux autres packages. Ainsi, le principe SDP est respecté, car X et Y dépendent respectivement d'un package plus stable qu'eux, d'où la direction de dépendance vers la stabilité.

Inversement, et d'après la *figure1.5.5(b)* nous pouvons considérer que le package A est très instable[AK06], car il dépend de deux autres package (X,Y). Ce qui fait que des éventuelles modifications sur les packages X et$Y$ entraineront des changements dans le package A. Ceci est fort probable dans la pratique, vu que ces deux packages peuvent être des sources externes (API, Bib,etc.).

Supposons que nous voulons ajouter un quatrième package dans la *figure1.5.5(b)*. Si ce package dépendra du package A, l'erreur sera fatale dans le cadre d'un processus de maintenance, car le principe SDP ne sera pas respecté dans ce cas.

9. ***Depend in the direction of stability.***

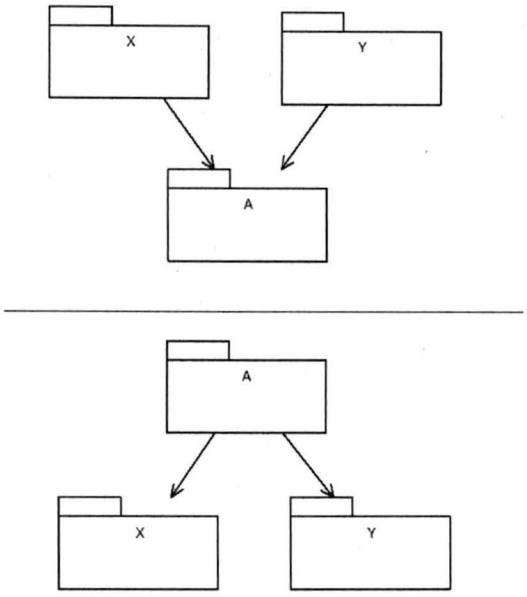


FIGURE 1.5.5 – Package Stable VS Instable inspiré par [AK06]

1.5.2.3 Stable Abstractions Principle : SAP

Un package devrait être aussi abstrait que stable [10].

Le principe SDP que nous avons présenté dans la section précédente, est une arme à double tranchant. À force d'appliquer ce principe, le modèle OO risque d'être de moins en moins flexible. En effet, un package stable est celui qu'on ne peut pas changer facilement. Or, avoir beaucoup de packages stables dans un modèle OO risque de bloquer les concepteurs qui ne pourront plus avoir assez de flexibilité pour étendre et améliorer leurs modèles. Ceci est un inconvénient majeur du principe SDP.

Pour remédier à cette problématique, R. Martin a introduit la notion d'abstraction en parallèle avec la notion de stabilité des packages. Le principe SAP est un principe qui vise à assurer une certaine équivalence entre le rapport abstraction et stabilité dans les packages d'un système OO. En effet, avoir des packages avec un certain niveau d'abstraction donne au modèle plus de flexibilité.

De plus, il ne faut pas se focaliser sur l'un des deux critères dans la conception d'un système

10. ***Stable packages should be abstract packages***

OO, sinon, on risque de se retrouver soit avec un modèle rigide, soit avec un modèle instable et très compliqué à maintenir. Pour conclure, Le principe SAP s'assure que n'importe quel package stable dans le modèle est assez abstrait pour ne pas impacter la flexibilité du modèle.

1.6 Les métriques de packages

Nous avons présenté dans la section précédente, l'ensemble des principes mis de l'avant pour avoir un modèle OO de bonne qualité. Cependant, ces principes restent des notions abstraites, difficiles à cerner en terme de valeurs quantitatives. Afin de s'assurer de l'application de ces principes sur un modèle quelconque, il faudra définir des métriques qui ciblent chacun de ces principes et retournent des valeurs quantitatives que nous pouvons manipuler et interpréter plus facilement.

Il est clair que le critère abstrait des packages en tant que composant nous impose de descendre d'un niveau pour définir les métriques de qualité de ce composant. Dans ce contexte, nous faisons référence aux entités qu'un package regroupe. En effet, la définition de n'importe quelle métrique d'évaluation qualitative ou quantitative d'un package sera appliquée au niveau des entités internes du package. La cohésion, le couplage, la stabilité et l'abstraction sont les dimensions les plus utilisées dans le domaine de l'évaluation de la qualité des packages, en particulier durant les processus de maintenance des applications OO[ADSA09, BBD+10, Mar05].

Généralement, le couplage et la cohésion sont les deux axes de base qui permettent aux mainteneurs d'évaluer la qualité des packages en termes de structure interne et externe. En effet, l'évaluation des packages à l'interne, considère les dépendances entre les entités internes du package. D'un autre côté, l'analyse externe s'applique en évaluant les dépendances du package par rapport au reste des packages dans le système OO. D'autre part, la stabilité et l'abstraction sont considérées comme des dimensions complémentaires. Elles visent à évaluer la flexibilité et la stabilité du modèle face aux changements qui peuvent s'appliquer lors d'un processus de maintenance ou de mise à niveau.

1.6.1 La cohésion

Une métrique de cohésion vise à évaluer le niveau de l'application du principe de cohésion CCP dans un modèle OO. En effet, et comme nous l'avons précisé dans le chapitre précédent, plu-

sieurs définitions ont été proposées pour cette métrique dans les études concernant le domaine du contrôle de la qualité.

D'après l'étude que nous avons faite, par rapport à la cohésion des packages, il n'y a pas de consensus clair pour la définition d'une telle métrique. Certains considèrent que la métrique de cohésion ne doit pas être mesurée qu'à partir de la seul structure interne du package, mais elle doit aussi prendre en compte les dépendances externes du package.

De toute évidence, nous avons conclu que la définition de chaque métrique dépend toujours des choix que l'évaluateur fixe dès le début. De plus, l'environnement de l'exécution et la nature des applications à analyser ont un impact majeur sur ces choix.

Lors de la réflexion par rapport à cette métrique, nous avons été obligés de faire des choix très importants concernant la définition de cette métrique. Dans ce qui suit, nous allons présenter la définition adoptée pour la mesure de la cohésion, ainsi que sa formule, suivie d'une section de réflexion, par rapport à la métrique de cohésion, illustrée à travers un exemple concret.

Nom : Cohésion (Package)

Acronyme : Cohesion

Références : [ADSA09]

Formules :

$Cohesion(P) = \frac{|DEP_{int(p)}|}{|DEP_{total(p)}|}$ *avec p est le package visé par la métrique.*

$DEP_{int(p)} = \sum|DEP_{int(i)}| \,/\, \{i \in [C1 \ldots Cn]\}$, $[C1 \ldots Cn]$ *est l'ensemble des classes du package p.*

$DEP_{total(p)} = \sum|DEP_{int(i)}| + \sum|DEP_{ext(i)}| \,,/\, \{i \in [C1 \ldots Cn]\}$

Définition : La cohésion d'un package P, est l'ensemble de dépendances internes de P par rapport à l'ensemble de dépendances totales de P.

Réflexion :

La définition d'une métrique efficace dépend toujours des critères que nous voulons mesurer. Pour cela, il faudra fixer dès le début les points que nous voulons évaluer.

La métrique de cohésion que nous définissons devra retourner des mesures qui varient de 0 à 1, où le 1 indique que le package a un taux de cohésion maximum. En effet, ces mesures nous aident à comprendre la structure interne d'un package, et à détecter s'il nécessite une

restructuration ou non. Or, dans notre travail, un package nécessite une restructuration lorsqu'il a un faible taux de cohésion. Par conséquent, nous pouvons déduire que certaines entités dans ce package sont mal positionnées.

La métrique de cohésion ne doit pas se limiter au package visé par la métrique, mais doit aussi prendre en compte ses dépendances externes. Afin de mieux comprendre l'idée derrière ce choix, prenons l'exemple ci-dessous, des deux figures (a) et (b), qui représentent une modélisation très simple d'un système OO avec deux packages.

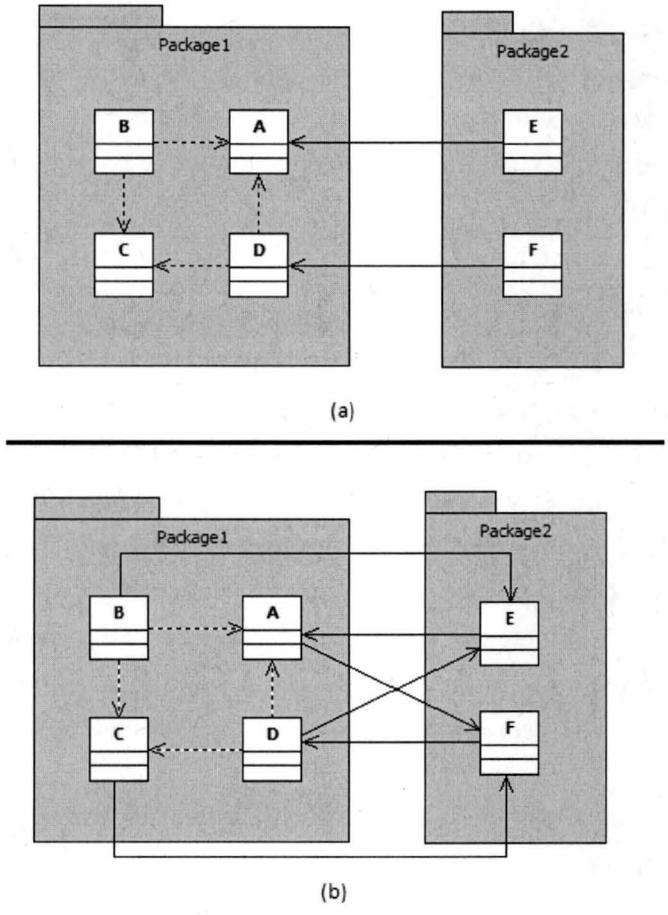


FIGURE 1.6.1 – Métrique de cohésion

Nous présentons dans la *figure1.6.1(a)*, une configuration de deux packages 1 et 2 où chaque package est différent de l'autre d'un point de vue structure. Le package1 contient quatre entités

(A, B, C, D). Ces entités sont reliées entre-elles via des dépendances internes. D'autres part, le package2 contient deux entités (E, F). Ces deux entités ont une relation de dépendance interne entre elles. Enfin, les deux packages ont des dépendances externes qui les relient l'un à l'autre via leurs entités respectives.

Sur un premier plan, si nous voulons étudier la cohésion du package1, on remarque que chaque entité de ce package est reliée avec une autre entité dans le même package. D'après le principe CCP, ce package doit avoir un bon taux de cohésion vue que toutes les entités dépendent entre elles, ce qui implique que n'importe quel changement dans l'une de ces entités, implique le changement des autres entités de ce package. Nous pouvons conclure que ce package peut avoir un taux de cohésion maximale. Or, nous assumons que ceci est insuffisant, dans la mesure où notre analyse ne s'est fixée que sur le package en question sans prendre en compte le reste des packages. En effet, nous avons mentionné précédemment que le package1 a des dépendances externes avec le package2 où $F \rightarrow D$ et $E \rightarrow A$. Ces dépendances peuvent être nuisibles dans le cadre d'un processus de restructuration. Nous devons donc prendre en compte ces dépendances aussi dans la définition de la métrique de cohésion.

Les entités qui sont impliquées dans des dépendances externes dans un package, sont appelées les entités d'interfaces[Mar05]. D'après la figure (a), A et D sont deux entités d'interface du package 1. Leurs dépendances n'ont pas été prises en compte par notre analyse au début. Ceci est une grave erreur, vu que n'importe quel changement au niveau de E et F aura des répercussions sur les entités du package 1. Certes, dans ce cas les changements seront mineurs par rapport à ceux des dépendances internes, vu que leur nombre est inférieur et ne concerne que les entités d'interfaces. Dans ce cas, le principe CCP n'est pas respecté en totalité. De plus, ceci ne répond pas aux critères que nous voulons mesurer à travers cette métrique :

- Identification des entités qui représentent une anomalie en terme de structuration interne des packages.
- Avoir une idée claire sur le rôle du package dans le système à travers la séamntique interne qui relie les entités du même package.
- Comprendre la structure interne du package.

Sur un deuxième plan, nous faisons une comparaison entre les figures (a) et (b) afin d'extraire les points essentiels qui influenceront notre choix pour la définition de la métrique. Dans la figure (b), nous avons gardé la même structure interne des deux packages. Cependant, nous avons augmenté les dépendances externes qui relient les entités d'interface de chaque package.

Par conséquent, nous concluons que la cohésion du package 1 dans la figure (b) va se dégrader par rapport a celle de la figure (a). En effet, le nombre de dépendances externes a augmenté, et même dépassé le nombre de dépendances internes du package. Donc, il est évident que les mesures qui seront retournées par la métrique doivent indiquer que ce package a un faible taux de cohésion, ce qui remettra en question l'emplacement de ces entités, et indiquera aux concepteurs que ce package est lié au package 2.

Le choix que nous avons adopté pour la définition de la métrique de cohésion ne couvre pas le cas où un système OO ne contient qu'un seul package. Les mesures, dans ce cas, ne sont pas applicables.

Pour conclure, nous avons choisi d'adopter une définition basée sur un raisonnement globale et non pas local, dans le but de détecter les anomalies reliées à la structuration des packages, ainsi que leurs relations dans un modèle OO. Nous jugons que la cohésion d'un package ne doit pas tenir compte que de ses dépendances internes, mais également, de ses dépendances externes. Autrement dit, la cohésion doit impliquer la totalité de dépendances du package.

1.6.2 Le couplage et ses dérivées

Une métrique de couplage vise à évaluer l'application du principe CRP [11] sur un modèle OO. En effet, cette métrique permet de mesurer le nombre de dépendances d'un package avec le reste des packages d'un système OO. Le couplage est complémentaire à la cohésion ; il permet d'évaluer la qualité de la modularisation d'un système OO. Dans la littérature, plusieurs définitions ont été proposées comme métriques de couplage. En revanche, ces définitions posent toujours des problèmes d'imprécision à cause de la nature abstraite du concept de couplage.

Un système OO de bonne qualité du point de vue de la modularisation, a des packages qui ne sont pas fortement liés les uns aux autres. Par conséquent, la métrique de couplage que nous devons définir doit retourner des mesures assez précises pour qu'on puisse juger si un package à un niveau de couplage satisfaisant ou pas. Ainsi, nous pourrons conclure si ce package nécessite une restructuration ou non.

Si dans un programme OO, un changement au niveau d'un package entraine des modifications sur un autre package, alors nous pouvons conclure qu'un couplage existe entre ces deux packages.

11. http ://en.wikipedia.org/wiki/Package_Principles

Il existe deux types de couplage, le couplage efférent et le couplage afférent.

Nom : Couplage Efférent (Package, Classes)

Acronyme : Ce

Références : [Mar94]

Formule :

$M = \{P1...Pn\}$; l'ensemble des packages du système.

$Ce(P_i) = |P_i \rightarrow P_j|$; $P_i \in M$, $P_j \in M$, $(i \neq j)$

Définition : le couplage efférent d'un package P, noté $Ce(p)$, est le nombre de packages dont P dépend.

Réflexion :

La mesure du couplage efférent d'un package passe par les classes. En effet, le *Ce* d'un package est le nombre de classes internes du package qui ont des dépendances sortantes externes avec des classes appartenant à un package différent.

Nom : Couplage afférent (Package, Classes)

Acronyme : Ca

Références : [Mar94]

Formule :

$M = \{P_1...P_n\}$; l'ensemble des packages du système.

$Ca(P_i) = |P_j \rightarrow P_i|$; $P_i \in M$, $P_j \in M$, $(i \neq j)$

Définition : le couplage afférent d'un package P, noté $Ca(p)$, est le nombre de packages qui dépendent du package P.

Réflexion :

La mesure du couplage afférent d'un package est l'inverse de celle du couplage efférent. En effet, $Ca(p)$ est le nombre de dépendances externes entrantes vers les classes du package visé par la métrique.

La volatilité des packages dans les systèmes OO est un problème assez fréquent dû à plusieurs facteurs. Le principal facteur est les interdépendances des packages dans un modèle OO, autrement dit, c'est l'ensemble des couplages externes des packages ($Ce + Ca$).

Le couplage des packages, représenté par les deux types de couplage que nous avons vus dans la section précédente, pose une problématique vis a vis du principe SDP [12]. En effet, un package avec un taux de *Ce* très élevé est un package instable qui risque de dégrader la qualité du modèle OO en le rendant très rigide. Cependant, l'inverse, un package avec un taux de *Ca* très élevé est un package très stable qui n'est pas facile à modifier. Or, un package avec un taux de *Ce* et *Ca* très élevés est encore pire, il est considéré comme un package très volatile, difficile à identifier comme étant un package stable ou instable.

Dans le but d'adresser ce problème, une métrique est définie afin de quantifier l'instabilité des packages dans un modèle OO.

Nom : Instabilité (Package)

Acronyme : I(P)

Références : [Mar94]

Formule :

$I(p_i) = \frac{Ce(p_i)}{Ce(p_i)+Ca(p_i)}$; $P_i \in M = \{P_1 .... P_n\}$

Définition : l'instabilité d'un package P, est le nombre de couplage efférent du package par rapport à la somme du couplage efférent et afférent.

Réflexion :

La mesure de la stabilité est une mesure qui se base sur l'ensemble des *Ce* et *Ca* d'un package. Cette mesure varie de 0 à 1, où $I(p) = 0$ indique que le package est complètement stable. Ainsi, I=1 indique que le package est instable. Le principal rôle de cette métrique est d'évaluer

12. http ://c2.com/cgi/wiki ?StableDependenciesPrinciple

la difficulté que risque de rencontrer les mainteneurs dans le cadre d'un processus de mise à niveau vis à vis de chaque package du modèle.

La mesure de la stabilité des packages est très efficace dans le sens où elle permet d'identifier les packages qui sont les plus difficiles à modifier dans un modèle OO. De plus, elle donne une idée claire sur la structure interne des packages, tout comme les relations qui les relient les uns aux autres. Cependant, l'application du principe SDP tend à rendre un modèle OO de plus en plus stable. Par conséquent, nous obtenons un modèle avec une architecture modulaire claire et simple à maintenir. Or, ceci peut être nuisible dans la mesure où le modèle devient de plus en plus fermé envers d'éventuelles extensions. En revanche, cela n'est pas forcément le but, ni un des critères fixés par les concepteurs pour un modèle de bonne qualité.

Le principe SAP est un principe complémentaire au principe SDP. En effet, ce principe vise à rendre les packages stables, les plus abstraits possible, pour permettre des extensions sur ce genre de packages. La métrique que nous allons présenter dans ce qui suit, vise à évaluer le niveau d'abstraction d'un package dans un système OO.

Nom : Abstraction (Package)

Acronyme : A(P)

Références : [Mar94]

Formule :

$A(p) = \frac{N_a}{N_c}$

N_a : Nombre des classes abstraites dans le package p.

N_c : Nombre total des classes du package p.

Définition : l'abstraction d'un package c'est la densité des classes abstraites à l'intérieur de ce package.

Réflexion :

Cette métrique permet de mesurer le niveau d'abstraction d'un package. En effet, ce n'est qu'une mesure de la densité des classes abstraites dans un package. Par conséquent, il faut mesurer le nombre des classes abstraites ainsi que le nombre total des classes. La mesure de

l'abstraction d'un package, varie de 0 à 1 également, où $A(p) = 1$, indique que le package ne contient que des classes abstraites et inversement pour $A(p) = 0$.

D'après notre analyse des métriques de l'instabilité et de l'abstraction, nous pouvons conclure qu'un certain équilibre doit être établit entre ces deux métriques. Autrement dit, dans les meilleurs des cas, un package a le même taux de stabilité et d'abstraction. Cependant, plus l'écart entre les deux métriques est grand, plus la qualité du package en termes de stabilité et d'abstraction est mauvaise.

Conclusion

La modularisation dans les modèles OO, est un attribut majeur pour l'évaluation de la qualité des systèmes OO. En effet, les interdépendances des modules, sont les facteurs principaux qui agissent, soit positivement soit négativement, sur la qualité des modèles OO. Certes, il est difficile de définir des métriques non ambigues pour mesurer les dépendances entre les packages à cause de leur nature abstraite. Ainsi, ces métriques ne peuvent être efficaces, que si nous avons un standard sur lequel nous pouvons nous référer pour leurs définitions.

Sur un premier plan, nous avons essayé de clarifier le plus possible la notion de la modularisation dans les systèmes OO à travers les packages. Tout d'abord, nous avons présenté les différents rôles qu'un package peut jouer dans un modèle OO. En effet, un package peut avoir plusieurs configurations possibles où chaque configuration reflète une sémantique précise, définie dès le départ, à l'étape de la conception d'un système OO. Également, nous avons analysé la notion de dépendance entre les packages où chaque dépendance a ses propres acteurs, les packages et les entités. Une relation de dépendance représente une source d'information utile pour notre processus d'évaluation de la qualité des modèles OO.

Les dépendances entre les packages ne sont pas toujours bonnes. Dans certains cas, elles représentent un inconvénient majeur qui dégrade la qualité des packages. Nous ne pouvons pas toujours avoir un jugement catégorique sur la qualité du package, à cause de la nature abstraite de ces composantes, et le rôle sémantique, qu'il joue dans les modèles OO. Les mesures retournées par les métriques de packages varient de 0 à 1. Par conséquent, nous pouvons toujours avoir une interprétation générique par rapport à l'ensemble du système, pour juger si un

package nécessite une restructuration ou non.

Sur un deuxième plan, nous avons présenté l'ensemble des principes que R. Martin a défini pour la conception des packages. Nous avons également présenté les métriques qui visent à mesurer le niveau de l'application de ces principes dans un modèle OO. Cependant, la définition présentée pour ces métriques reste de très haut niveau, et leur application réelle nécessite une approche précise et efficace pour avoir des mesures concrètes et efficaces pour un éventuel processus de restructuration. Nous discutons dans le chapitre suivant, les différentes approches et métriques proposées dans la littérature, pour l'analyse et la mesure de la qualité des packages dans les modèles OO.

Chapitre 2

État de l'art

2.1 Introduction

Nous allons présenter dans le présent chapitre une revue de la littérature qui reflète les différents travaux et réflexions qui ont été proposés dans le domaine de la mesure de la qualité des packages, autrement dit, la modularisation dans les systèmes orientés-objets. En effet, ce chapitre est très important pour notre travail de recherche, qui est basé sur la définition d'un ensemble de métriques qui permettent de mesurer les notions que nous essayons d'évaluer dans ce travail. Les études qui ont porté sur l'analyse des modèles orientés-objets sont très variées, car elles prennent souvent un angle assez particulier. Une analyse de la structuration des packages dans les systèmes OO, peut-être dynamique en portant sur les environnements d'exécution, ou statique en se focalisant sur une analyse statique des modèles orientés-objets.

Il est évident que les trois critères d'analyse cités dans le chapitre précedent sont très différents, mais cela n'empêche pas que plusieurs aspects techniques, telles que les métriques utilisées dans chaque critère, peuvent être très proches en terme d'appellation, avec des fondement théoriques et pratiques différents par rapport aux autres aspects dans d'autres critères d'analyse.

Tous au long de ce chapitre, nous allons présenter les métriques utilisées pour l'évaluation de la structure des packages dans les systèmes orientés-objets. En premier lieu, nous allons présenter les différentes études et recherches concernant la structure des packages. Ensuite, nous allons étaler les différents aspects théoriques et pratiques des métriques utilisées pour l'évaluation de la qualité des packages. Finalement, nous allons mettre en évidence, les principes de conception des packages dans les modèles OO définis par plusieurs chercheurs.

Dans le but de présenter une analyse complète des métriques définies pour les packages, nous allons présenter brièvement quelques métriques dynamiques. Nous jugeons que l'étude de l'aspect dynamique des mesures concernant les packages est nécessaire, cela nous permettra par la suite de bien argumenter les choix et les politiques utilisés pour l'évaluation des packages dans les modèles OO.

2.2 Les principes de conception des packages

La nature abstraite des packages rend le processus d'évaluation quantitative assez dificile. Mais cela n'empeche pas qu'une bonne définition de la structuration modulaire dans les modèles

Principes	Description
REP	Release Equivalency principle : une bonne structuraration d'un package comprend des classes qui sont réutilisables ensemble
CRP	Common Reuse Principle : les classes qui ne peuvent pas être réutilisées ensemble ne doivent pas être dans le même package
CCP	Common Closure Principle : les classes qui changent ensemble, vont ensemble
ADP	Acyclic Dependencies Principle : les dépendances entre les packages ne doivent pas fournie un cycle
SDP	Stable Dependencies Principle : un package qui recoit beaucoup de dépendances entrantes est un package stable

TABLE 2.2.1 – Description des principes de conception des packages

OO, permet d'avoir des mesures efficaces et précises. C'est pour cela que plusieurs travaux de recherche ont essayé de définir des principes de conception permettant de prédire la qualité des modèles OO.

Le terme principe de conception a été beaucoup utilisé dans la littérature. Ce terme, a été introduit par R.Martin dans son analyse des patrons de conception et leur impact sur la qualité des modèles OO. En effet, la mise en place d'un processus automatisé pour la vérification de la qualité des modèles OO, nécessite également des principes de haut niveau sur lesquels on peut se référer, au niveau des mesures.

Les principes de conception des packages ont été discutés principalement dans le travail de Martin[Mar03]. Le tableau *2.2.1* présente brièvement une déscription de chacun de ces principes.

D'autre part, Szyperski propose un principe qui se base sur la distinction entre les stratifications strictes et non-strictes. les stratifications strictes s'imposent lorsque une couche ne dépend directement que de la couche en dessous. Cependant, dans la stratification non stricte, une couche peu utiliser n'importe qu'elle couche en dessous[Szy02].

2.3 Analyse et représentation de dépendances

Dans la présente section, nous décrivons les différents travaux de recherche concernant la détection de la dépendance inter et/ou intra packages. En effet, dans la première partie, nous

allons décrire les différentes approches et méthodologies d'un point de vue haut niveau. Dans la deuxième partie, nous allons détailler les différentes techniques de détection et de mesure, adoptées pour l'évaluation qualitative des packages.

Dans la littérature, les approches adoptées pour la collection des données quantitatives et qualitatives sont très variées. Abdeen et al. [ADSA09]présentent un algorithme optimisant la structure modulaire dans un programme orienté-objets, en se basant sur des données quantitatives collectées à partir du code source. Ces mesures manquent de précision dans la mesure où le processus se base sur le parsing direct du code source. De même, une mauvaise structuration ne peut être détectée que lors d'une phase très tardive du processus de développement. Par conséquent, une éventuelle restructuration des packages risque d'être coûteuse, au même niveau que la maintenance.

Dans le but d'adresser ce problème, d'autres travaux ont tenté d'appliquer des métriques sur les modèles OO afin de détecter les mauvaises structurations des packages durant la phase de conception. À titre d'exemple, Godfrey et al. [DG09] ont essayé de présenter une nouvelle approche qui utilise les techniques de rétro-ingénieurie [1] pour élaborer un modèle hybride modélisant la collaboration et la similarité entre les packages. La détection de dépendances inter-packages dans cette approche, est réduite à l'ensemble des messages transmis entre les packages durant l'exécution. Visiblement, les résultats s'avèrent précis mais leur efficacité est remise en cause, car les mesures collectées sont limitées, vu qu'elles ne représentent qu'un seul plan d'exécution. En revanche, ces mesures ne peuvent pas être exploitées en tant que mesures de référence sur la quelles on peut se baser pour proposer une meilleur structuration des packages.

Nous ne pouvons pas parler des mesures de conception niveau packages dans les systèmes OO, sans se préoccuper des mesures de conception niveau classes. En effet, R. Martin propose dans son livre [Mar03], des principes de conception niveau classes, qui diffèrent légèrement des principes qu'il présente pour les packages. Mais ce qui est évident pour nous, c'est que les mesures proposées pour la conception des classes et des packages, sont des mesures complémentaires dans le sens où les classes sont la base des packages, d'un point de vue conceptuel. Par déduction, chaque composant a ces propres principes de conception indépendamment de leurs fortes liaisons conceptuelles.

Dans le même sens, Adrian.M et al. [MP05] proposent des mesures conceptuelles niveau

1. http ://fr.wikipedia.org/wiki/R%C3%A9troing%C3%A9nierie

classes, mais qui prennent en compte l'aspect sémantique qui relie les classes sans se limiter aux informations structurelles que les classes partagent. L'idée de représenter les similarités sémantiques entre les composants d'un modèles OO, a été aussi introduite par Godfrey et al. dans leurs mesures conceptuelles des packages[DG09]. Malheuresement, le critère abstrait de ces informations qui ne dépendent que du programmeur en question, dégrade le niveau de précision de ces mesures.

2.4 Les métriques statiques

L'évaluation de la qualité de la structuration des packages dans les modèles OO, a poussé les chercheurs à définir plusieurs métriques. En effet, ces métriques permettent d'avoir des mesures quantitatives et qualitatives des packages. Leurs principaux rôles sont de vérifier numériquement le niveau de respectabilité des principes de conception qui définissent la bonne structuration des packages dans les systèmes OO. Malgré la variété des métriques proposées dans la littérature, il est clair qu'un consensus de haut niveau s'est imposé par rapport à ces métriques, à savoir, le couplage et la cohésion. De plus, d'autres métriques telles que la stabilité et l'abstraction sont définies comme des mesures complémentaires [Mar94].

Dans ce qui suit, nous allons présenter en premiers plan les métriques de nature quantitative. Par la suite, nous discutons les métriques de nature qualitative proposées dans la littérature.

2.4.1 Mesure de qualité (mesure de dépendances)

Le couplage et la cohésion sont les principales mesures définies dans la littérature comme des métriques d'évaluation de la qualité des modèles OO, en terme de structuration modulaire. Cependant, leurs définitions de base diffèrent d'un travail à un autre. En effet, le couplage est traduit par les dépendances inter-modulaires. Or, la cohésion est traduite par les dépendances intra-modulaire. Goulâo M et al.[eAG01] proposent une restructuration modulaire qui se base sur l'analyse de la similarité des classes par paires. À cet égard, le couplage est catégorisé en douze catégories qui représentent les affinités possibles entre les paires. Ensuite, des nouveaux clusters [2] regroupent les classes en maximisant les affinités intra-modulaires et en minimisant les affinités inter-modulaires.

2. http ://en.wikipedia.org/wiki/Cluster_(computing)

Chitra.S et al. [AK06] considèrent un module comme une collection d'éléments de traitement qui interagissent pour avoir une sortie quelconque. La cohésion selon eux, est définie par l'ensemble des éléments du même module qui dépendent entre eux. D'autre part, le couplage est défini par le nombre de messages transmis d'un objet à un autre, à condition que ces deux objets ne soient pas dans le même module. On remarque d'après ces deux définitions, que la cohésion est définie par le nombre d'éléments actifs dans le module par rapport au nombre total des éléments du module. Ceci implique que contrairement à la définition du couplage, la fréquence de connexion entre deux objets ou plus, n'est pas considérée dans la métrique de cohésion. Par conséquent, les mesures établies par cette métrique, ne reflète pas le niveau de cohésion réelle du module.

En ce qui concerne le couplage, Chitra S et al.[AK06] affirment que l'augmentation des interconnexions complexes du module, implique un modèle OO strictement couplé. Ainsi, la qualité du modèle se dégrade remarquablement. La définition du couplage présentée n'est pas assez précise dans la mesure où la vraie question qui doit se poser, est qu'est-ce qu'on entend par inter-connexion complexe ? Il faut bien préciser la définition de cette métrique. Dans notre cas [AK06], Chitra S et al. proposent de définir l'inter-connexion du couplage en terme d'invocation de méthodes et d'instanciation d'objets dans d'autres types d'objets. Ceci ne couvre pas les différents cas possible, car elle ne prend pas en compte les éventuelles variations de la notion de dépendance entre deux classes en fonction de la nature du langage OO utilisé pour développer le logiciel en cours d'analyse.

Kamalraj et al. [KKRH10] proposent une méthodologie qui consiste à exploiter des scripts de tests dans le but d'identifier les packages qui sont susceptibles d'être réutilisables. Dans leur approche, ils utilisent des références numériques basées sur des heuristiques obtenues à partir de plusieurs métriques. Ils présentent la cohésion comme l'ensemble des liens internes dans un package, et le couplage comme l'ensemble des liens entre packages, en proposant le chiffre 20 comme seuil de tolérance pour le nombre de liens inter-packages. Le fait de ne pas bien détailler la nature des liens, et de se baser sur un seuil fixe qui n'est pas flexible en fonction de la nature ou de l'ampleur du programme, peut mettre en cause la crédibilité des résultats obtenus.

Sous un autre angle de vue, la cohésion selon T. Zahou et al. [ZXS+08] doit prendre en compte l'aspect contextuel du package. En effet, la majorité des métriques de cohésion proposées, tel que c'est mentionné précédemment, ne prend en compte que les dépendances intra-packages

pour mesurer la cohésion. Or, ceci peut provoquer des défauts au niveau de la métrique. En effet, la cohésion basée sur le contexte est définie par l'ensemble de données qu'un composant partage avec d'autres composants dans le même package et inversement. Ceci reprend l'idée qui consiste à mesurer l'ensemble des messages transmis entre les composants.

Wile et al. [WH92] proposent dans leurs travaux, une analyse approfondie de la problématique reliée à la liaison dynamique et aux différents types de dépendances possibles entre les objets dans le paradigme OO. En effet, ils catégorisent les dépendances comme suit :

- Les dépendances de données (entre deux variables)
- Les dépendances d'invocations (entre deux méthodes)
- Les dépendances fonctionnelles (entre deux modules)
- Les dépendances de définitions (entre une variable et son type)

Nous considérons que les auteurs ont couvert la majorité des aspects de programmation qui peuvent engendrer une dépendance entre deux objets. Cependant, ces définitions manquent de précision, par exemple, d'un point de vue spécification de langage, par quoi se traduit la notion de la dépendance fonctionnelle entre deux modules, et est-ce que les variables locales (niveau méthodes) et globales (niveau classes) sont considérées comme dépendances de données ?

Hani Abdeen et al. [Abd09] distinguent deux approches pour définir la cohésion. Ils estiment que la cohésion peut être traitée sur une échelle réduite, au niveau du module, où elle représente l'inter-connexion des classes au sein d'un module. Autrement, la cohésion peut être définie sur une échelle plus large, niveau système, où on considère que la liaison de deux classes internes avec un module commun, représente une relation conceptuelle entre ces deux classes. Concernant le couplage, il est défini comme suit : un package P réfère à un package Q, implique qu'il existe au minimum une classe du package P qui réfère à une classe du package Q.

Le couplage, en général, est considéré comme un ensemble de dépendances entre les classes de deux modules différents. À cet égard, Balmas.F et Briand.LC [BBD+10, BWL99], détectent les dépendances entre les classes en terme d'invocation de méthodes. Mais, ils ignorent le polymorphisme, puisque selon eux, le polymorphisme est une caractéristique naturelle dans le paradigme OO. Elle n'est pas considérée comme une forme de dépendance entre les classes.

Manifestement, la cohésion selon R.Martin [Mar94], repose en grande partie sur la notion de réutilisabilité. En effet, un package est complètement réutilisable s'il répond aux trois critères définies par Booch [3] dans sa définition des *Class Catégorie*.

3. http ://en.wikipedia.org/wiki/Grady_Booch

- Règle 1 : n'importe quelle modification sur une classe de l'ensemble des classes au sein d'une catégorie, entraine une modification de toutes les autres classes.
- Règle 2 : si une classe de l'ensemble est réutilisée, cela entraine la réutilisation de tout le reste des classes.
- Règle 3 : les classes d'une catégorie partagent un ensemble de fonctions communes.

Dans le cadre de l'évaluation qualitative des packages, H Melton et al. [MT07] ont défini une nouvelle métrique qui s'appelle "CRSS" [4]. Elle permet de mesurer la qualité conceptuelle des packages. En effet, Cette métrique permet de vérifier si les relations entre les classes dans un système OO, l'empêche d'avoir une bonne conception de packages. Techniquement, cette métrique vérifie pour une classe donnée, sa relation avec le reste des classes, dans le code source, d'un point de vue unité de compilation. Du coup, cette technique a l'avantage de gérer les dépendances par rapport aux classes de tout le système, et non pas avec les classes d'un sous-système (package).

D'un point de vue conceptuel, un package est un mécanisme qui permet d'organiser les classes dans des *namespaces* [5]. Par conséquent, nous estimons qu'une étude de la littérature concernant les définitions proposées pour la cohésion niveau classe s'impose. La cohésion dans sa définition de base décrit les liaisons des éléments à l'intérieur d'un même objet [MP08]. La majorité des études définit la cohésion des classes en tant que référencement des attributs d'un objet dans les méthodes du même objet. Certains proposent une analyse sémantique additionnelle du code source pour mesurer le niveau de cohésion d'une classe [MP05].

2.4.2 Mesures complémentaires

Dans le cadre de l'évaluation des modèles OO, plusieurs métriques complémentaires ont été définies en s'inspirant des principes de conception des modules. Ces métriques permettent d'évaluer la strucuration des packages. En effet, elles permettent aux maintaineurs, d'évaluer la performance du modèle en termes de modification et de réutilisabilité.

Instabilité/Abstraction

Cette métrique vise principalement à mesurer l'instabilité des packages dans les OOD [6]. R.Martin [Mar94] qualifie une classe de stable lorsque cette dernière ne dépend de rien d'autre. Dans le

4. Class Reachability Set Size
5. http ://en.wikipedia.org/wiki/Namespace_(computer_science)
6. Oriented-Object Design

même sens, une classe est fortement responsable lorsque elle a un grand nombre de classes qui dépendent d'elle. Elle à la responsabilité d'être le plus stable possible pour ne pas rendre le modèle fragile, ce qui affecterait systématiquement la crédibilité du modèle d'un point de vue maintenance. En effet, la stabilité excessive des packages dans un modèle OO affecte la flexibilité du modèle. Donc, pour une structuration modulaire optimale, il faut avoir quelques modules flexibles dans les OOD. Par conséquent, la notion de l'abstraction dans le processus d'évaluation des modèles OO s'impose.

D'autre part, Chitra.S et al. [AK06] définissent la métrique de stabilité suivant deux angles de vue. Le premier est en terme de dépendance, où on considère que la stabilité est reliée à la quantité de travail nécessaire pour faire les modifications requises sur un package. Pour cela, l'augmentation du nombre de dépendances entrantes du package augmente sa complexité, ainsi que sa stabilité. En ce qui concerne le deuxième angle de vue, la stabilité peut être perçue en terme d'abstraction. En effet, dans le but d'éviter le fait d'avoir un modèle figé à cause de la stabilité de dépendances dans un modèle OO, l'augmentation du niveau d'abstraction permet d'améliorer la flexibilité du modèle. Ainsi, on obtient un package stable fortement abstrait.

Sur un autre plan, établir un équilibre entre les deux principes SDP et SAP nécessite une métrique qui vérifie numériquement les limites respectées par les packages, par rapport à ces deux principes. R.Martin [Mar94] propose de créer un graphe qui combine les deux métriques, instabilité et abstraction en tracant une ligne droite entre les deux points de coordonnées, ($I = 1$, $A = 0$) et ($I = 0, A = 1$). L'analyse de ce graphe, engendre la création d'une nouvelle métrique qui mesure la distance des packages par rapport à cette droite appelée, séquence principale (*Main Sequence*). Elle représente la ligne de la zone d'équilbre entre l'abstraction et l'instabilité que l'on désire avoir pour un package (voir *figure 2.4.1*).

Les dépendances cycliques

Le principe de dépendance cyclique discuté par R.Martin [Mar03], est un principe dérivé de la notion de dépendance inter-packages. En effet, son implication flagrante dans la dégradation de la qualité des modèles OO, exige du développeur une métrique qui permet de mesurer le niveau de couplage cyclique de son modèle. À cet égard, beaucoup de travaux ont essayé de définir des métriques permettant de mesurer le niveau acyclique d'un système OO. Par exemple, H.Abdeen et al. [ADSA09] le définissent comme le ratio de dépendances sortantes qui déclenchent le

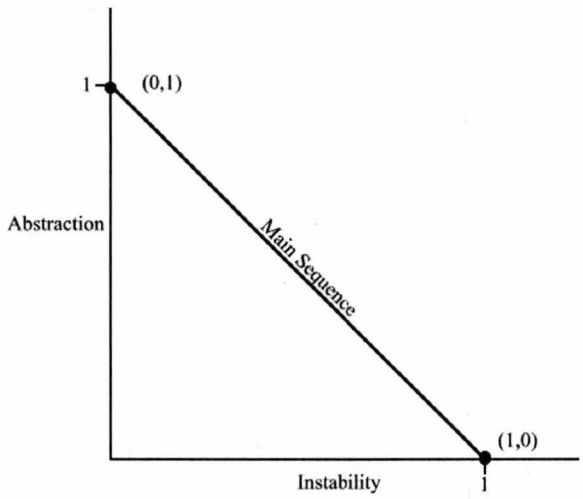


FIGURE 2.4.1 – Graphe I vs A[Mar94]

cycle entre deux packages, par rapport à l'ensemble des dépendances sortantes de tout le système. Cependant, ils catégorisent ces dépendances en deux types, à savoir, les *dépendances cycliques inter-packages directes* (les dépendances entre classes de deux packages différents et qui formant un ou plusierus cycles de dépendances), et les *connexions cycliques inter-packages* (nombre des packages impliqués dans un ou plusieurs cycles de dépendances).

Cette classification n'est pas utile, dans la mesure où une dépendance cyclique entre deux classes appartenant à deux packages différents, entrainera systématiquement une connexion cyclique entre ces deux packages. Par conséquent, la métrique de connexion entre les packages n'est pas très efficace. En effet, le nombre de packages impliqués dans la dépendance cyclique peut être déterminé à partir de la métrique de dépendances cycliques directes. Le principe de dépendance acyclique *ADP,* est très difficile à maintenir. E Hautus [Hau02] propose un outil de réingénierie des packages participant à des dépendances cycliques. Il crée des couches abstraites pour ces packages, mais en ignorant les dépendances cycliques entre elles. En effet, ces dépendances sont tellement courantes et facile à créer durant la phase de développement, qu'il est presque impossible de les éliminer au complet.

Goulao et al. [eAG01] utilisent des métriques à valeurs quatitatives, dans le but d'analyser la structuration des packages, d'une manière plus technique. Selon eux, une approche quantitative vise à répondre aux questions suivantes :

1. Quel est le nombre optimal de modules pour un système OO ?

2. Quelle est la taille moyenne optimale des modules dans un système OO ?
3. Comment répartir d'une manière efficace les classes d'un système OO, sur ses modules ?

En plus des métriques de couplage et de cohésion, Goulâo M et al. ont défini la métrique RMD (Relative Module Dispersion) qui permet de mesurer la dispersion relative des classes dans les modules d'un système OO. Cette métrique utilise une autre métrique, appelée AMM (Average Module Membership), qui représente la moyenne du nombre de classes dans un module. Bien que ces métriques sont utiles pour l'évaluation de la qualité d'un module en terme de taille optimale, mais elles ne sont pas assez précises dans la mesure où on peut pas les considérer efficaces dans tout les cas. De plus, d'après notre étude dans ce domaine, les relations de dépendances des modules dans les système OO, est la problématique qui préoccupe souvent les développeurs et les mainteneurs.

2.5 Les métriques dynamiques

Dans la partie suivante nous allons présenter une revue de la littérature sur les métriques dynamiques définies pour les packages. Généralement, les mesures dynamiques sont utilisées pour le contrôle de la qualité des nouvelles techniques d'optimisation proposées, en particulier pour les compilateurs et les machines virtuelles [DDHV03, SYA, WH92]. Pendant longtemps, les recherches destinées à définir des métriques pour le paradigme OO se sont basées sur des mesures et des analyses statiques. Mais l'augmentation remarquable de la complexité des programmes OO a touché à la crédibilité des résultats retournées par les métriques statiques. En effet, le principal inconvénient des métriques statiques du point de vue des concepteurs et des développeurs, est l'inefficacité. Les résultats retournés par ces métriques ne peuvent pas s'appliquer sur des mesures en relation avec l'environnement d'exécution. La nécessité de définir des métriques dynamiques s'est donc imposée.

Le couplage, la cohésion et la complexité sont les principales métriques dynamiques proposées dans la littérature. En effet, la complexité dynamique du comportement des applications, et la mesure reportée sur les aspects des programmes OO étaient les principales motivations derrière l'idée d'utiliser des métriques dynamiques plutôt que des métriques statiques.

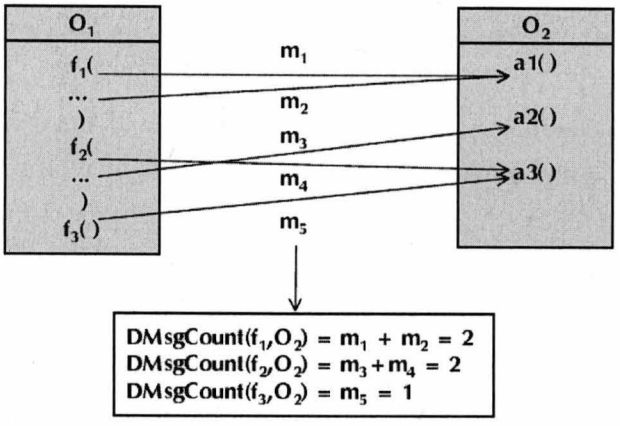


FIGURE 2.5.1 – Le couplage dynamique [CKKR98]

2.5.1 Le couplage dynamique

Le couplage dynamique est une métrique définie par le nombre de messages échangés entres les objets d'une application au moment de l'exécution. Nous pouvons donc déduire que le couplage dynamique est une mesure qui se fait au niveau des objets, alors que le couplage statique tel qu'il est défini, mesure le couplage au niveau classe, ce qui rend la mesure beaucoup moins précise.

E.Cho et al. [CKKR98] représentent le couplage dynamique par le total des nombres de chargement des messages échangés entre chaque fonction d'un objet avec chaque fonction d'un autre objet. Et pour mieux illustrer notre définition de cette métrique, nous présentons la figure ci-dessous. D'après la *figure 2.5.1*, le couplage dynamique est mesuré en fonction du nombre de messages échangés entre l'ensemble des fonctions de l'objet O_1 et l'ensemble des fonctions de l'objet O_2. Cependant, il est important de prendre en compte, que lors de l'exécution, les mesures relatives aux chargements des messages échangés diffèrent d'un message à un autre.

Ainsi, le couplage dynamique ne repose pas que sur le nombre de messages échangés, tel que c'est le cas avec le couplage statique. En effet, le couplage dynamique a l'avantage de définir les mesures relatives aux chargements du trafic des messages, ce qui nous permet d'identifier quelles sont les méthodes les plus souvent invoquées.

D'un autre coté, le couplage dynamique nous permet de quantifier la taille des messages char-

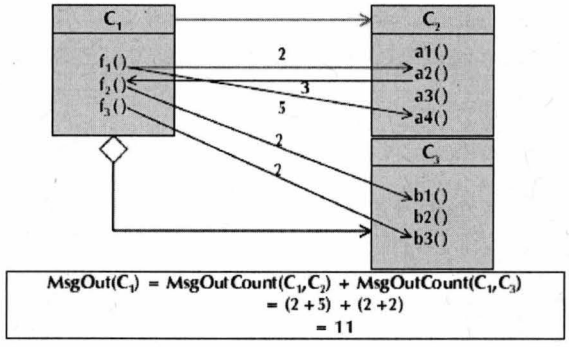


FIGURE 2.5.2 – Représentation graphique de la cohésion [CKKR98]

gés [7], qui est défini par le temps de réponse d'une fonction suite à son invocation. Cette métrique peut être très utile dans le cas où nous voulons faire une amélioration sur la structuration de l'application analysée. En prenant l'exemple de la *figure 2.5.1*, la taille de chargement entre l'objet O_2 et la fonction f_1 est égal à la somme des temps de réponses de l'ensemble des fonctions de O_2 lorsqu'elles invoquent la fonction f_1 de l'objet O_1. À titre d'exemple, la taille du couple (f_1, O_2) est égale aux temps de réponse de $m_1 + m_2$. Ceci nous permet de calculer la taille de chargement entres les deux objets O_1 et O_2, définie par la somme des tailles de chargement de chaque fonction de O_1 avec O_2.

2.5.2 La cohésion dynamique

D'un point de vue général, la cohésion est la mesure complémentaire du couplage. C'est une métrique qui mesure la cohésion entre les classes. Son but est de quantifier le nombre d'appels vis à vis des fonctions d'une classe par apport aux autres classes, mais dans le même objet.

D'un point de vue statique, la cohésion est le nombre total des messages passés d'une fonction à une autre sans compter la valeur du poids des messages transférés.

D'un point de vue dynamique, la cohésion est une mesure qui permet d'avoir des données quantitatives sur l'aspect dynamique interne d'un objet précis. En effet, c'est le nombre de messages transférés plus la taille de chargement de chaque message [CKKR98].

Afin de mieux comprendre la définition de la cohésion dynamique, nous décrivons la *figure*

7. Message LoadSize [CKKR98]

2.5.2 ci-dessus. D'après cette figure, C_1, C_2 et C_3 appartiennent au même objet O_1. D'après E.Cho et al. [CKKR98], la cohésion dynamique de O_1 est le nombre de messages sortants vers les autres classes plus la taille de chargement de chaque message (le nombre dynamique des messages). Cependant, la taille de chargement entres deux objets est la somme de tous les chargements des messages, tel que cela est expliqué dans la section sur le couplage dynamique.

2.5.3 La complexité dynamique

La mesure de la complexité dans les applications orientées-objets est très importante d'un point de vue qualitatif. En effet, la complexité a été définie depuis 1976 par la complexité cyclomatique de McCable [8]. La complexité de McCable mesure la complexité d'un programme en fontion du graphe de flot de contrôle (control flow graph) qui représente les différents chemins possibles d'un programme durant l'exécution. La formule suivante $VG = e - n + 1$ permet de calculer le nombre de chemins possibles durant l'exécution, ce qui représente la complexité dynamique d'un programme durant l'exécution. Dans notre cas, l'application pratique de la théorie de ce graphe, implique que la variable e (le nombre d'arêtes) peut représenter le nombre d'invocation de méthodes d'un objet à un autre, et n (nombre des nœuds) représente les modules ou les objets dans un programme. Par exemple, Yacoub et al. [SYA] définissent la complexité de McCable en se basant sur la spécification de ROOMchart [9], qui est un langage de modélisation pour les systèmes à temps réel. En effet, ROOMchart est consitué de plusieurs sous-ensembles de transitions primitives. Ils définissent une transition primitive par une relation entre deux états. Une transition est un *trigger* [10]qui se déclenche suite à une ou plusieurs conditions, pour l'exécution d'un ensembles d'actions. Par analogie avec le graphe de calcul de la complexité de McCable, les arêtes sont représentées par le flux de contrôle d'une transition et les segments représentent le code d'action ou de décision. Les nœuds sont représentées par les points d'entrée/sortie du programme. La *figure 2.5.3* illuste le graphe de flux de contrôle pour une transition primitive.

Beaucoup d'études s'intéressaient au concept de la complexité des objets, mais comme mesure statique, car la définition et l'implémentation des métriques de complexité dynamique, en fonction de la théorie de graphe de McCable, n'est pas une tâche facile et les résultats ne sont pas

8. http ://en.wikipedia.org/wiki/Cyclomatic_complexity
9. Real-time Oriented Object Modeling.
10. Déclancheur de la transition de l'état d'entrée vers l'etat de sortie

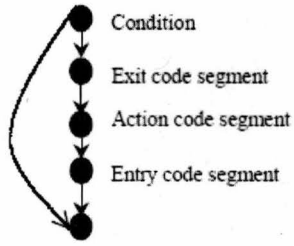


Figure 2.5.3: Flux de contrôle pour une transition primitive [SYA]

garantis.

Chidamber et al. [CK94] ont défini des métriques tels que la profondeur de l'arbre de l'héritage (DIT [11]), le nombre de fils (NOC [12]), ou la complexité pondérée des méthodes (WMC [13]), comme mesures pour l'évaluation de la complexité d'un modèle OO d'un point de vue conceptuel. D'autre part, Henry et al. [LH93] définissent la complexité de chaque objet par leurs nombres d'attributs et de méthodes. Toutes ces variations au niveau de la définition concernant la complexité des programmes OO, en termes de conception ou d'implémentation, témoignent de la difficulté de cette tâche.

Conclusion

Nous avons présenté dans ce chapitre l'état de l'art des mesures adoptées dans le processus d'évaluation de la qualité des packages dans le paradigme de l'orienté-objets. En effet, les approches et les techniques d'évaluation varient suivant l'angle de vue sur le quel on se réfère pour définir les métriques d'évaluation. Il est clair que les outils d'évaluation, en particulier, ceux de la rétro-ingénierie, sont les plus sollicités par les contrôleurs de la qualité, vu qu'ils permettent d'avoir des mesures dans une phase précoce du processus de développement, à savoir, la phase de conception.

Sur un autre plan, nous avons également présenté l'état de l'art des métriques dynamiques adoptées dans la littérature, pour analyser les interractions et les relations des objets dans un

11. http ://maisqual.squoring.com/wiki/index.php/Depth_in_Inheritance_Tree
12. http ://maisqual.squoring.com/wiki/index.php/Number_of_Children
13. http ://maisqual.squoring.com/wiki/index.php/Weighted_Methods_per_Class

système OO. Ces mesures sont totalement différentes des mesures statiques présentées dans la première partie. Elles ont la spécificité de mesurer les interractions entre les objets selon un plan partiel du système OO, mais cela reste de loin beaucoup plus précis que les mesures statique. Les métriques dynamiques étaient rarement étudiées dans la littérature. D'ailleurs, c'est un sujet de discussion assez récent.

Chaque technique de mesure a ses avantages et ses inconveinients. Par conséquent, c'est le domaine de l'utilisation de ces mesures qui impose le choix de la nature des métriques à utiliser. À titre d'exemple, les métriques dynamiques sont le plus souvent utilisées dans les compilateurs et les machines virtuelles, où on a souvent besoin de métriques applicables dans un environnement d'exécution, pour optimiser les performances de ces outils. Cependant, les métriques statiques sont le plus souvent utilisées dans les outils de rétro-ingénierie par les contrôleurs de la qualité. En effet, ce genre de métriques est le seul moyen qui produit des mesures assez génériques, pour former un consensus d'évaluation applicable sur tous les projets orientés-objets.

Ainsi, nous avons choisi de travailler avec les métriques statiques, parceque l'une de nos contributions dans ce travail de recherche, est la définition de métriques précises, flexibles et surtout applicables à tous les programmes orientés objets. Nous présentons dans le chapitre suivant notre approche conceptuelle pour la mise en œuvre de notre processus d'évaluation modulaire des systèmes OO.

Chapitre 3

Génération du méta-modèle

Le nombre de langages OO présents sur le marché s'est multiplié durant la dernière décennie d'une façon remarquable. À cet effet, les développeurs utilisent de plus en plus les modèles, vu que cela leur permet de représenter les systèmes OO suivant des vues différentes, indépendamment de la plate-forme de développement.

Un modèle est une représentation graphique et textuelle d'un système réel. En effet, il permet de structurer les informations relatives à un système, suivant trois couches (données, traitement et flux de données). Cependant, dans certains cas, les modèles ne sont plus assez flexibles pour mettre en œuvre le point de vue ou la théorie que les développeurs souhaitent modéliser. D'où, la notion des méta-modèles.

Dans la littérature, un méta-modèle est une modélisation d'un niveau d'abstraction plus élevé que celui des modèles. Il permet de définir la structure d'autres modèles. En effet, le principal avantage des méta-modèles, est qu'ils donnent aux développeurs la possibilité de définir des modèles suivant un point de vue abstrait ce qui les rend indépendants du langage sur lequel ils sont définis. Ainsi, on pourra les interpréter par des langages de programmation différents.

Actuellement, les méta-modèles sont de plus en plus adaptés aux plate-formes de calcul des métriques. En effet, dans ce domaine, la précision et la clarté de la définition des métriques sont deux critères primordiaux pour la réussite de n'importe quel processus d'évaluation. Or, en pratique, ce n'est pas souvent le cas. Brian et al. considèrent que les métriques peuvent être interprétées de plusieurs façons, en particulier lorsqu'il s'agit d'évaluer une notion abstraite, telle que c'est le cas pour la cohésion et le couplage dans l'analyse de la dépendance entre les modules et les classes[Mar03, MP08].

Dans le but d'adresser le problème d'ambiguïté, plusieurs chercheurs ont eu recours à la notion des méta-modèles. Brian et al. utilisent une présentation canonique pour modéliser les métriques de couplage et de cohésion. D'autre part, Mens et Lanza proposent un méta-modèle pour le calcul des métriques dans les programmes OO, mais ils ne considèrent pas les métriques de couplage et de cohésion dans leurs travaux [ML02].

Dans le même sens, nous allons présenter dans la première section de ce chapitre, les applications des méta-modèles dans le domaine du calcul des métriques. Ensuite, dans la section 2, nous présentons en détail la plate-forme d'extraction des métriques dans laquelle nous avons intégré nos extensions pour le méta-modèle existant. Dans la section 3, on décrit le méta-modèle de la plate-forme avant les extensions. Finalement, la section 4 décrit les extensions que nous

avons apportées au niveau du méta-modèle existant, pour la modélisation de dépendances entre les packages et les classes.

3.1 Méta-modèles et métriques

L'utilisation des méta-modèles pour la définition et le calcul des métriques est l'un des moyens les plus efficace utilisé pour définir des métriques claires non ambiguës, ce qui permettra d'avoir des outils de mesure efficaces et flexibles. Dans ce qui suit, nous allons présenter en premier lieu, la définition des méta-modèles d'un point de vue théorique. Ensuite, nous étudions la possibilité d'adapter cette notion au domaine de la mesure des programmes OO.

3.1.1 Définition

Un méta-modèle représente généralement un point de vue particulier, par rapport au système que l'en souhaite établir. D'après la *figure3.1.1* ci-dessous, on distingue deux entrées nécessaires pour la définition d'un méta-modèle.

Tout d'abord, on définit le point de vue comme étant l'élément de départ. Il peut-être représenté suivant une théorie ou une conception particulière par rapport aux informations que nous voulons représenter du côté du code source. Ensuite, la création du méta-modèle est exprimée moyennant un langage ou un formalisme de conception. En effet, il existe plusieurs langages permettant de définir un méta-modèle. Parmi les langages les plus utilisés, on trouve le langage MOF [1]. Le deuxième point d'entrée sont les métriques. Elles seront définies suivant le méta-modèle, ce qui permet d'avoir plus de flexibilité et de précision par rapport aux mesures retournées par ces métriques.

3.1.2 Utilisation des méta-modèles pour la définition de métriques

Dans la littérature, plusieurs métriques ont été proposées dans le but d'évaluer la qualité des programmes OO. Cependant, l'ambiguïté et les différentes interprétations possibles de ces métriques ont été les principales problématiques de ce domaine. Par exemple, Churcher et Sheperd

1. MOF : http ://fr.wikipedia.org/wiki/Meta-Object_Facility

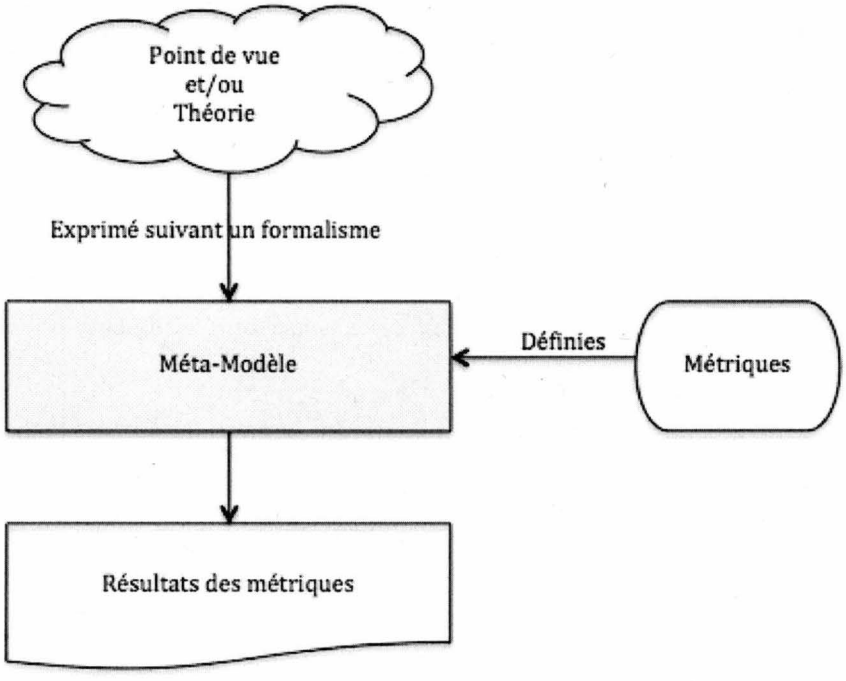


FIGURE 3.1.1 – Méta-modèle adapté au domaine de calcul des métriques

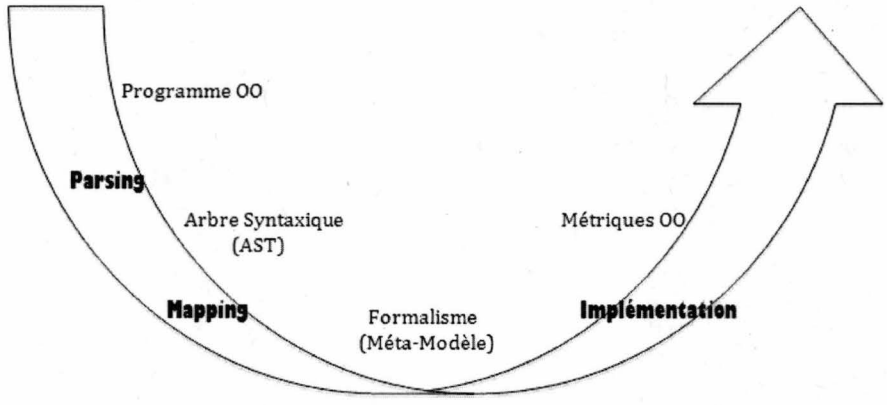


FIGURE 3.1.2 – Architecture d'un outil de calcul des métriques basé sur les méta-modèle

ont démontré qu'il existe une ambiguïté par rapport à la suite des métriques proposées par Chidamber et Kemerer, concernant le couplage et la cohésion dans les programmes OO [Bas95].

Certains chercheurs estiment que l'utilisation d'un méta-modèle peut résoudre le problème d'ambiguïté des métriques. En effet, les méta-modèles jouent le rôle d'une source de données utilisée par les fonctions de calcul des métriques.

L'avantage de cette approche est qu'elle permet de récolter les informations que l'on juge pertinentes pour le calcul des métriques. Ainsi, le processus de calcul ne dépend plus du code source, mais du méta-modèle en question. De plus, cette approche donne plus de flexibilité dans la mesure où elle permet de créer un méta-modèle particulier suivant un point de vue élaboré par les développeurs, dans le but de mieux répondre aux fonctions de calcul des métriques.

La *figure3.1.2* représente la transition entre les couches d'une plate-forme de calcul des métriques basée sur les méta-modèles. Dans un premier lieu, nous avons le code source du programme OO ciblé par l'analyse. Dans la phase de *parsing*, nous appliquons un parseur sémantique sur l'arbre syntaxique pour récolter les informations nécessaires pour le calcul des métriques. Ensuite, ces informations sont modélisées durant la phase de *mapping* suivant un formalisme représenté par la notion de méta-modèle. Finalement, les mesures seront implémentées en se basant sur le méta-modèle généré.

En revanche, nous précisons que le formalisme peut être retiré de l'architecture, ce qui rendra la phase d'implémentation des métriques dépendante de l'arbre syntaxique. Ainsi, leurs des-

criptions dépendront directement du code source. Or, c'est justement cela l'erreur que nous voulons éviter.

3.2 Présentation de la plate-forme d'extraction des métriques

Notre travail se situe dans le cadre d'une plate-forme d'extraction de métriques basée sur un langage de description de haut niveau. La plate-forme a été implémentée par AliKacem et Sahraoui sous forme d'un plug-in eclipse[AS09]. En effet, ils utilisent les techniques de rétro-ingénierie pour établir un méta-modèle générique. Ce modèle contient toutes les données nécessaires pour le calcul des métriques.

La plate-forme est composée de deux processus complémentaires. Au premier plan, nous avons le processus de parsing et mapping. Au deuxième plan, nous avons le processus d'extraction des métriques. Dans ce chapitre, nous allons nous contenter de présenter le premier processus ainsi que le module que nous avons intégré dans ce processus. Le deuxième processus sera détaillé dans le chapitre suivant.

3.2.1 Architecture de la plate-forme

La *figure3.2.1* décrit l'architecture générale de la plate-forme. D'après la figure, on distingue deux sous-systèmes : un pour la représentation du code source et l'autre pour la collection des métriques[AS09].

Tous d'abord, le module de "*parsing et mapping* " assure la représentation du code source en méta-modèle. Dans ce contexte, nous précisons que ce n'est pas l'intégralité du code source qui est représenté. En effet, la modélisation ne concerne que les notions nécessaires pour le calcul des métriques.

Par la suite, le module de collection des métriques prend la relève et utilise le méta-modèle établi à la phase précédente, comme source de données pour calculer les métriques. Également, ce module comprend un langage de description de métriques de haut niveau. Il permet de définir un ensemble de métriques suivant une syntaxe simple, que l'on expliquera en détail au chapitre suivant. Le module évaluateur joue le rôle de l'interpréteur du langage, il interprète

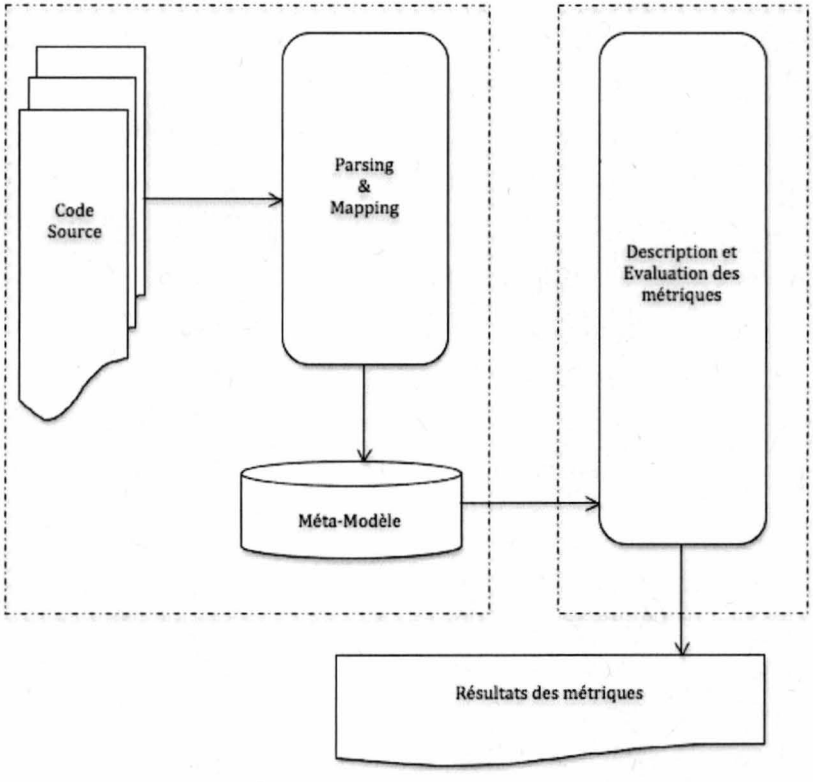


FIGURE 3.2.1 – Architecture de la plate-forme inspiré de [AS09]

les métriques définies et réalise les calculs relatifs à ces métriques. Finalement, les résultats des calculs sont extraits vers l'interface de la plate-forme.

3.3 Un méta-modèle pour la mesure des programmes OO

3.3.1 Motivation

Depuis que les systèmes OO sont de plus en plus complexes, la qualité et la maintenance de ces systèmes sont devenues une priorité pour les développeurs. À cet effet, plusieurs métriques ont été définies dans le but de quantifier leur qualité. Souvent, les métriques sont définies à partir du code source. Or, ceci n'est pas pratique, dans la mesure où la métrique dépend du langage de programmation. Par conséquent, la métrique ne peut pas être appliquée sur un autre programme développé avec un langage de programmation différent.

De plus, de nos jours, les programmes OO sont souvent composés d'un ensemble de sous-systèmes qui sont fortement connexes. De même, les métriques sont de plus en plus difficiles à définir. Ainsi, on se retrouve avec des métriques ambiguës, non efficaces et ouvertes à plusieurs interprétations. L'utilisation d'un méta-modèle qui agit comme une couche d'isolation entre la métrique et le code source, résout le problème de la mobilité et de la clarté des définitions des métriques.

L'utilisation des métas-modèles pour la définition des métriques permet de les rendre plus claires. Ainsi, nous espérons arriver à un consensus au niveau de l'interprétation de ces métriques. La conception d'un méta-modèle générique pour la plate-forme d'extraction de métriques que nous avons présentée dans la section précédente est loin d'être une tâche facile. En effet, le méta-modèle doit gérer les différents aspects des langages de programmation OO. Bien que la majorité de ces langages partagent les mêmes concepts, ils peuvent partager la même syntaxe mais la sémantique peut être différente.

3.3.2 Vue d'ensemble du méta-modèle adopté pour la plate-forme

Généralement, l'utilisation des méta-modèles dans les plate-formes de calcul de métriques consiste à récupérer les données à partir du code source pour les représenter sous la forme

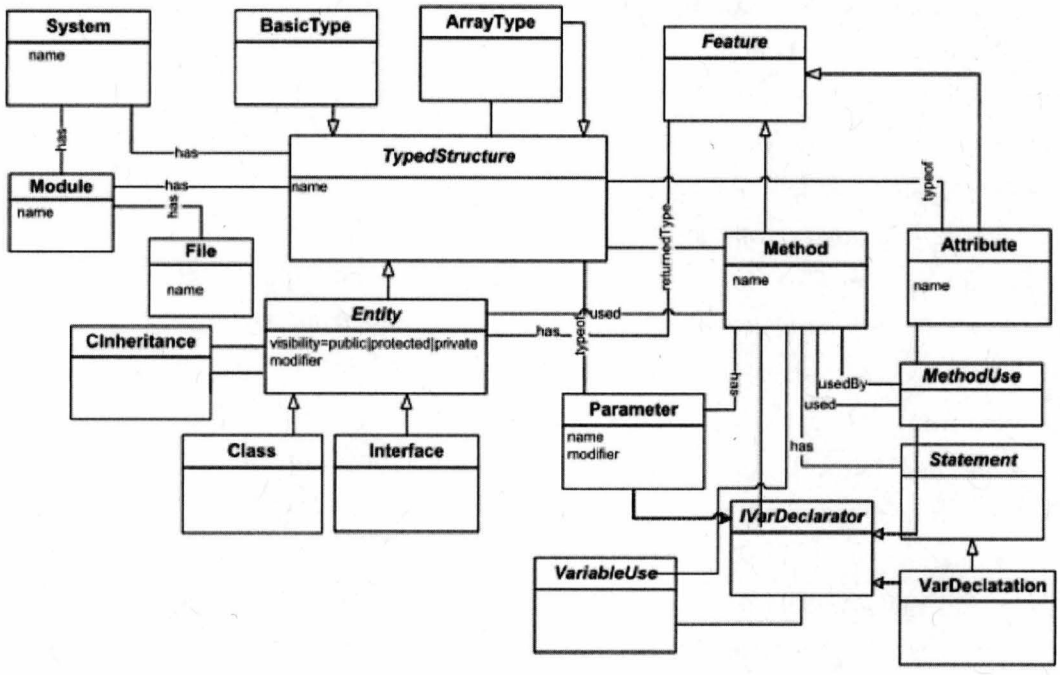


FIGURE 3.3.1 – Vue d'ensemble partielle du méta-modèle de la plate-forme de calcul et d'extraction de métriques[AS09]

d'un méta-modèle, ceci, dans le but de faciliter leur manipulation. Alikacem et Sahraoui ont conçus un méta-modèle générique qui supporte les programmes OO dans différents langages de programmation. La *figure3.3.1*, représente une vue partielle du méta-modèle utilisé pour la plate-forme d'extraction de métriques.

Tel que nous pouvons l'apercevoir dans la *figure3.3.1*, le méta-modèle contient les éléments de base d'un programme OO. Ils gèrent tous les aspects des langages OO. En effet, pour avoir un modèle générique applicable sur les programmes OO développés avec différents langages de programmation, Alikacem et Sahraoui ont étudié les concepts communs et non communs qui pourront influencer l'intégrité des méta-modèles par rapport au code source d'origine dans la plate-forme.

- **Les concepts communs :** sont ceux où les langages OO partagent la syntaxe et la sémantique, tels que les classes, les attributs et les méthodes. Ces composantes sont représentées dans le modèle ci-dessus par les *ClassDef*, *Attribut* et *Method.*
- **Les concepts variables :** sont ceux où les langages OO partagent la même syntaxe, mais pas

la sémantique. L'héritage et le polymorphisme sont des exemples typiques pour ce genre de concepts. En effet, la problématique reliée à la sémantique adoptée par les différents langages OO pour l'héritage, est contournée en représentant ces concepts lors du processus de *parsing et mapping* du code source. Tous les attributs et les méthodes hérités sont dupliqués dans les sous-classes suivant la sémantique du langage.

- **Les concepts spécifiques :** sont des concepts qui n'existent que dans des langages particuliers tels que *Java* et *C++*. Ils sont représentés explicitement dans le méta-modèle. Par exemple, la notion des entités est une notion abstraite présente dans ces langages. Elle joue le rôle de la classe mère dans le méta-modèle qui généralise les différents types d'entités que l'on peut trouver. D'après la *figure3.3.1*, les deux éléments *Class* et *Interface* sont deux spécialisations de l'élément *Entity*. En revanche, si nous voulons adopter d'autres types d'entités nous pouvons étendre le modèle en ajoutant le type adéquat comme spécialisation de l'élément *Entity*.

La génération du méta-modèle suivant ces trois concepts, permet d'avoir un modèle générique assez complet qui englobe les aspects syntaxiques et sémantiques d'un programme OO. Les deux premiers concepts sont utilisés pour mesurer les métriques. Cependant, le troisième concept est représenté dans le modèle d'une façon abstraite, mais utile pour calculer certaines métriques telles que le nombre d'entités lorsque la nature de l'entité n'est pas considérée dans la définition de la métrique [AS09].

Comme nous l'avons démontré précédemment dans la *figure3.2.1*, le module de *parsing* et *mapping* est le module responsable de la génération du méta-modèle dans la plate-forme.

Dans le but de tester l'efficacité de la plate-forme sur un langage en particulier, le module *parsing* et *mapping* doit être implémenté suivant ce langage. Dans ce sens, Alikacem et Sahraoui ont implémenté ce module suivant le langage Java dans le but de tester les limites du méta-modèle dans un environnement Java. En effet, ils assument que l'adaptation du méta-modèle sur d'autres langages nécessitera des modifications mineures vu que le modèle gère les trois concepts décrits précédemment.

3.4 Extension du méta-modèle

3.4.1 Mise en contexte

Nous avons présenté dans la section précédente une vue d'ensemble de la plate-forme d'extraction de métriques implémentée par Alikacem et al., ainsi que le méta-modèle adopté pour cette plate-forme. Dans le but d'intégrer notre module de calcul de métriques pour l'analyse et la mesure de la dépendance des packages dans les programmes OO, des extensions au niveau du méta-modèle s'imposent. Ceci est loin d'être une tâche triviale. En effet, les extensions apportées sur le modèle doivent être génériques et applicables sur tous les langages OO. Ainsi, il faut prendre en compte tous les concepts que nous avons définis précédemment pour la conception d'un méta-modèle générique.

Les différentes étapes que nous avons suivies pour établir les extensions du méta-modèle sont les suivants :

1. L'identification des composantes du programme OO déclencheurs de la dépendance entre les packages au niveau du code source et le méta-modèle.
2. La modélisation de la notion de dépendances au niveau du méta-modèle.
3. L'implémentation des extensions apportées au niveau du module de *parsing* et *mapping*.

3.4.2 Détection de la dépendance entre les packages

Nous avons démontré dans le chapitre 2 que la dépendance entre les packages est mesurée à travers les dépendances entre les classes. En effet, puisque les packages sont de nature abstraite dans les langages OO, il est tout à fait légitime de se référer à un niveau d'abstraction plus bas que les packages. Généralement les classes, autrement dit les entités, sont les composantes concrètes d'un package au niveau du code source.

D'un point de vue statique, s'il existe une dépendance entre deux classes, cela signifie systématiquement qu'une dépendance entre leurs packages respectifs existe aussi. Ainsi, nous pouvons conclure que la dépendance est transitive entre les classes et les packages.

$$\exists(C_1 \rightarrow C_2) \Longleftrightarrow (P_1 \rightarrow P_2) \,/\, [C_1 \in P_1, C_2 \in P_2]$$

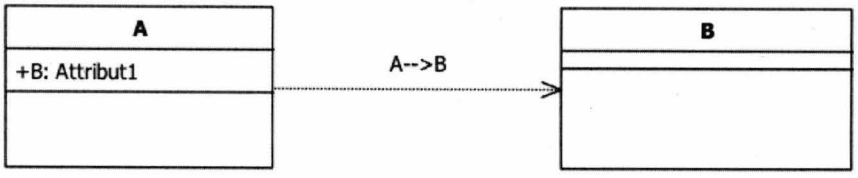


FIGURE 3.4.1 – Dépendance à travers la déclaration d'un attribut

S'il existe une relation de dépendance entre deux classes C_1 et C_2, cela signifie que leurs packages respectifs P_1 et P_2 sont en dépendance aussi, et inversement. À savoir, P_1 et P_2 peuvent être le même package lorsque il s'agit d'une relation de dépendance interne, ou deux packages différents lorsqu'il s'agit d'une relation de dépendance externe.

Pour établir les extensions nécessaires au niveau du méta-modèle de la plate-forme de calcul et d'extraction des métriques, nous devons identifier les éléments du modèle, qui sont susceptibles d'être les déclencheurs de cette dépendance. Or, vu que le méta-modèle n'est qu'une représentation générique du code source d'un programme OO, nous devons définir clairement les instructions qui peuvent déclencher une dépendance entre deux classes au niveau du code source.

D'après notre analyse statique des différents concepts des programmes OO, une relation de dépendance entre deux entités existe lorsque il y a :

– **Déclaration d'un attribut : (concept commun)**

La sémantique adoptée pour les attributs dans les langages OO est la même. Par conséquent, la déclaration des attributs est un concept commun qui ne demande pas un traitement de faveur pour sa modélisation. D'après la *figure3.3.1,* l'élément *Attribute* est en relation *typeOf* avec l'élément *TypedStructure* qui généralise les différents types que l'on peut avoir dans un programme OO[2]. Généralement, la déclaration des attributs se fait au niveau de l'entité. Or, si l'attribut est typé d'une autre entité définie dans le programme, alors une relation de dépendance existe entre l'entité qui contient la déclaration de l'attribut et l'entité cible.

La *figure3.4.1* représente d'une manière simple la relation de dépendance qui peux exister à cause de la déclaration d'un attribut. L'entité A contient la déclaration d'un attribut de type

2. BasicType : pour les types basiques (int, char, etc)
ArrayType : pour les tableaux
Entity : pour les types composés crées dans le programme.

B. Donc, statiquement A dépend de B ($A \rightarrow B$). À cet effet, l'élément *Attribut* acteur de la dépendance doit être pris en compte dans notre modélisation.

– **Déclaration d'une méthode : (concept commun)**

Les méthodes représentent une source importante de dépendance. En effet, la déclaration d'une méthode contient plusieurs éléments clefs déclencheurs de la dépendance entre les entités.

1. Type retourné par la méthode : la déclaration d'une méthode au sein d'une entité, peut retourner un objet typé d'une autre entité définie dans le programme. Ainsi, une relation de dépendance existe entre les deux entités.
2. Paramètre d'entrée de la méthode : la signature d'une méthode peut avoir une liste d'objets passés en paramètre d'entrée. Dans le cas où ces objets sont typés d'une ou plusieurs autres entités déclarés dans le programme analysé, une dépendance statique existe entre l'entité hôte de la méthode et les entités ciblées par les paramètres de la méthode. Ainsi, nous pouvons conclure que l'élément *Parameter* dans le méta-modèle présenté dans la *figure3.3.1* doit être pris en compte par notre modélisation de la dépendance.
3. Variable locale : le corps de la méthode peut contenir plusieurs variables locales. Ces variables peuvent être typés d'une autre entité dans le programme. Ainsi, on considère qu'une relation de dépendance existe entre l'entité en cours et l'entité cible.
4. Accès à un attribut : une méthode qui accède à un attribut appartenant à une autre entité, est considérée comme une forme de dépendance entre les deux entités. D'après le méta-modèle existant, l'élément *VariableUse* représente la liste des attributs accédés par la méthode.
5. Appel de méthode : l'appel de méthode est la forme la plus claire parmi les formes de dépendance entre les entités. En effet, dans la littérature, les appels de méthodes sont souvent considérés comme une forme de dépendance statique et même dynamique entre les entités[ADSA09, Gup11] . Les appels de méthodes sont représentés par l'élément *MethodeUse* dans le méta-modèle de la figure3.3.1. Cet élément et en relation avec l'élément *Method* qui représente la méthode qui contient l'appel de la méthode.

– **Une relation d'héritage entre deux entités : (concept variable)**

Dans le pradigme de l'orienté-objet, l'héritage est un concept variable, qui dépend du langage de programmation[AS09]. En effet, le méta-modèle concu pour la plate-forme supporte les différents types d'héritages. D'après la *figure3.3.1*, l'élément *CInheritance* est en relation reflexive avec l'élément *Entity*. Son rôle est de représenter les informations relatives à l'héritage

entre les différentes entités du programme analysé. Il contient deux attributs principaux, *Parent* et *Child,* qui représentent respectivement les entités mère et fille.

Tel que nous l'avons précisé dans la section précédente, les concepts variables sont représentés explicitement durant la phase de mapping. Ceci n'influence pas notre travail dans la mesure où nous jugeons que l'héritage entre deux entités signifie qu'une dépendance statique entre ces deux entités existe. Par conséquent, l'élément *CInheritance* doit être pris en compte dans notre modélisation de la dépendance.

– **Instructions : (concept spécifique)**

Au niveau du code source, il existe plusieurs types d'instructions qui représentent une forme de dépendance entre les entités. Certes, ces instructions diffèrent d'un langage à un autre, mais pour des raisons d'implémentation, nous avons considéré deux types d'instructions typiques du langage *Java* :

1. InstanceOf : le code source d'une entité peut contenir l'instruction *instanceOf* qui permet de vérifier le type d'un objet. Si l'objet testé est typé d'une autre entité, une relation de dépendance existe aussi entre les deux entités actrices.
2. Cast : le cast est considéré comme une forme de dépendance entre l'entité qui contient dans son code source cette instruction et l'entités visée par le *cast.*

– **Les classes imbriquées : (concept spécifique)**

Certains langages tels que Java et C++, supportent les classes imbriquées [3]. Or, une classe qui contient la déclaration d'une autre classe, est une forme plus forte que l'héritage entre deux classes. Par conséquent, les deux classes sont en dépendance statique entre-elles. Cela, n'est présent que dans certains langages OO, sa représentation se fait explicitement au niveau du méta-modèle.

Finalement, on considère que toutes les instructions décrites dans cette section représentent une forme de dépendance statique entre deux entités. Ces informations doivent être intégrées dans le méta-modèle existant sous forme de dépendance entre les entités. À savoir, étendre le méta-modèle pour supporter toutes ces formes de dépendances.

3. Nested Class

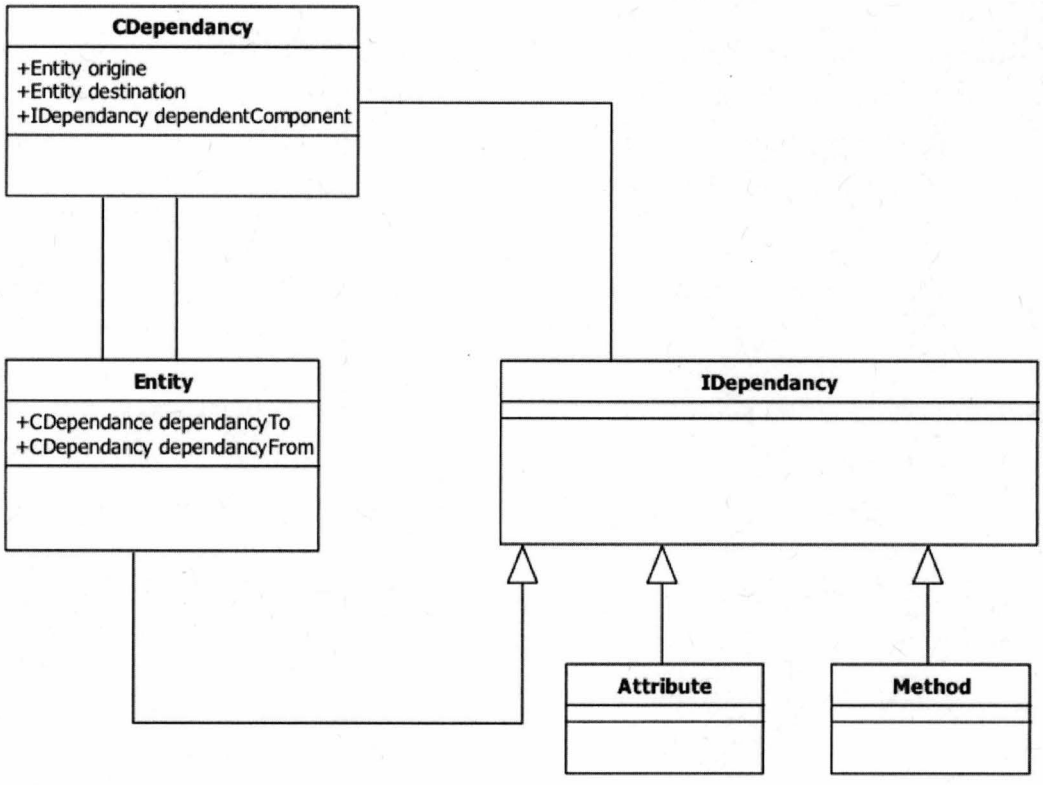


FIGURE 3.4.2 – Extension du méta-modèle

3.4.3 Extension du méta-modèle pour supporter les dépendances entre les entités

L'évaluation de la qualité des packages dans les programme OO, nécessite des extensions au niveau du méta-modèle actuel. En effet, nous avons étendu le méta-modèle pour représenter les informations relatives à la dépendance entre les entités, et par la suite, mesurer la dépendance entre les packages. La *figure3.4.2* représente une vue partielle des extensions que nous avons intégrées dans le méta-modèle de la plate-forme, pour représenter les dépendances entre les entités et les packages.

D'après la *figure3.4.2*, nous avons ajouté l'élément *CDependancy* qui contient toutes les informations nécessaires pour le calcul des métriques de dépendance. Cet élément a pour rôle de définir une relation de dépendance entre deux entités. Les attributs *origine* et *destination* font

référence à l'entité qui est à l'origine de la dépendance et l'entité ciblée par la dépendance.

D'autre part, l'élément *IDependancy* généralise tous les éléments susceptibles de déclencher une dépendance. Ainsi, lors du calcul des métriques, nous pouvons avoir accès aux éléments acteurs de la dépendance, ce qui nous donnera plus d'informations et de précision par rapport aux dépendances modulaires des programmes OO.

Conclusion

Les recherches portées sur les mesures dans le paradigme de l'orienté-objet sont en constante progression. Ceci est dû au fait que les métriques OO actuelles, ne sont pas assez satisfaisantes au regard des développeurs. En effet, l'augmentation de la complexité des programmes OO, impacte systématiquement la complexité des métriques OO. Ainsi, les outils de calcul des métriques actuels ne sont plus efficaces à l'égard de certaines métriques OO, tels que la cohésion, le couplage et la complexité.

Récemment, certains chercheurs[AS09, Mar03, DR11] ont proposé l'idée d'utiliser un formalisme tel que les méta-modèles dans les outils de calcul des métriques. L'intégration des méta-modèles dans les outils de calcul des métriques permet d'avoir plus de flexibilité et de clarté par rapport à la définition des métriques.

Dans ce chapitre, nous avons présenté la plate-forme de calcul et d'extraction des métriques, implémentées par Alikacem et Sahraoui. La plate-forme adopte un méta-modèle générique qui représente les différents concepts des programmes OO. Ce méta-modèle joue le rôle de la source des données sur laquelle on peut se baser pour définir des métriques claires, non ambiguës et assez flexibles pour supporter des éventuelles modifications.

Dans le cadre de notre travail d'analyse et de mesure de la qualité des packages dans les programmes OO, des extensions sur le méta-modèle actuel de la plate-forme s'imposent. À cet effet, nous avons défini de nouveaux éléments au niveau du méta-modèle afin de représenter les informations concernant la dépendance entre les entités et les packages, qui seront essentiels à notre implémentation des métriques visant à évaluer la qualité des packages.

Dans la littérature, un consensus s'est imposé par rapport aux métriques, permettant d'analyser la qualité des packages, tels que la cohésion, le couplage, la stabilité et l'abstraction. Ces métriques sont souvent critiquées comme étant ambiguës. Dans ce sens, nous allons essayer

d'implémenter ces métriques au sein de la plate-forme en nous basant sur le méta-modèle décrit précédemment. Cependant, et comme nous l'avons précisé au cours de ce chapitre, les dépendances entre les packages passent par les entités, et par conséquent, d'autres métriques de base relatives aux entités, doivent être implémentées.

Dans le chapitre suivant, nous allons présenter l'ensemble des métriques que nous avons implémentées pour notre module de calcul des métriques de packages.

Chapitre 4

L'intégration du module de calcul des métriques

Le deuxième volet de notre travail consiste à implémenter et intégrer un module de calcul des métriques. Le but est d'analyser la qualité des packages dans les programmes OO, indépendamment du langage de programmation. Le module doit mettre à la disposition des utilisateurs de la plate-forme un ensemble de métriques permettant d'analyser la qualité des packages d'une manière claire et non ambiguë.

Dans le chapitre précédent, nous avons préparé le terrain à travers les extensions apportées au méta-modèle. Le méta-modèle sera l'unique source des données sur laquelle nous nous baserons pour calculer nos métriques.

Notre module de calcul des métriques sera intégré dans le processus d'évaluation des métriques décrit dans la *figure4.1.1.* En effet, l'évaluateur joue le rôle d'un interpréteur pour un langage de description des métriques, implémenté au sein de la plate-forme[AS09].

Le langage est appelé PatOIS [1]. Il permet de décrire des métriques suivant une syntaxe simple et efficace. Nous allons présenter en premier lieu l'outil qui supporte ce langage. Dans la section 2 nous allons définir ses différentes structures syntaxiques et sémantiques. La section 3 décrit le module d'évaluation et la structure mise en place pour extraire les données du méta-modèle, afin d'isoler les procédures de calcul des métriques du méta-modèle. Finalement, nous définissons dans la section 4 les métriques que nous avons implémentées au sein du langage.

4.1 L'outil de description et d'extraction des métriques (Boap FrameWork)

4.1.1 Architecture de l'outil Boap Framework

La mise en oeuvre de l'approche que nous utilisons pour la définition des métriques à travers un méta-modèle, nécessite l'implémentation d'un outil de support pour la plate-forme. L'outil a pour but de faciliter l'interraction avec la plate-forme. À cet effet, l'outil *Boap FrameWork* est implimenté sous la forme d'un plug-in dans l'environement *Eclipse*. L'exploitation de la plate-forme se résume en deux étapes :

1. Description des métriques : l'outil joue le rôle de l'éditeur du langage de description des métriques (PatOIS).

1. Langage de description de métriques proposé par Alikacem et Sahraoui : [AS09]

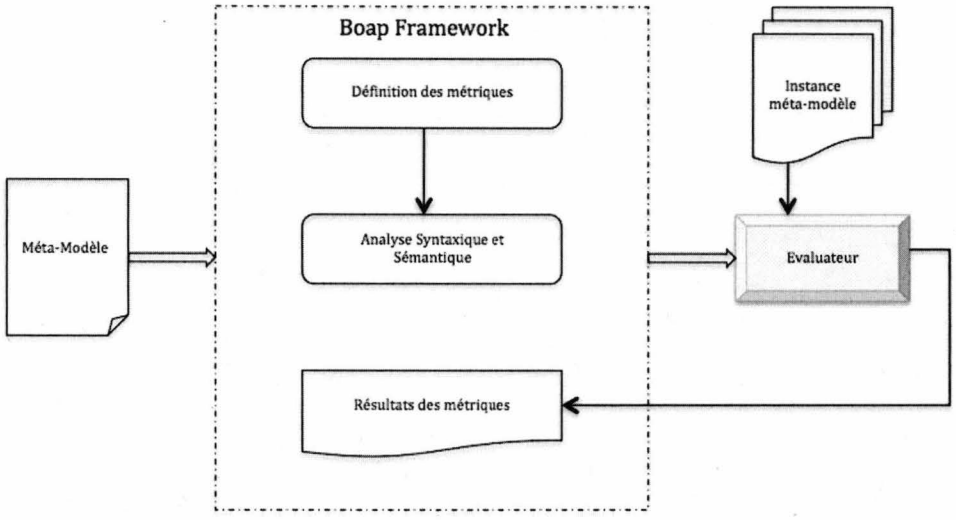


FIGURE 4.1.1 – Boap Framework : Architecture de l'outil de calcul des métriques dans l'environnement Eclipse.

2. Évaluation des métriques : l'outil effectue la vérification syntaxique et sémantique des métriques définies suivant le langage PatOIS. Par la suite, le module évaluateur effectue les calculs des métriques et retourne les résultats.

Afin de mieux comprendre le fonctionnement de l'outil, les entrées et les sorties sont représentées dans la figure ci-dessous.

L'outil *Boap FrameWork* permet l'utilisation de la plate-forme d'une manière conviviale. D'après la *figure4.1.1*, l'outil se déclenche dès que le processus de la rétro-ingénierie du programme OO est terminée. En effet, dès que le méta-modèle est établi, on commence par définir les métriques en fonction du méta-modèle, via le langage *PatOIS*. Ce langage contient un ensemble de métriques primitives que nous avons implémentées pour l'analyse de la qualité des packages.

L'analyseur syntaxique et sémantique vérifie la conformité des métriques définies par l'utilisateur en utilisant nos primitives suivant le langage *PatOIS*. Par la suite, le module Évaluateur prend la relève et envoie les requêtes adéquates à l'instance du méta-modèle créé. Finalement, les résultats de calcul des métriques sont retournés à l'outil *Boap Framework* où ils seront affichés.

FIGURE 4.1.2 – Prototype de l'outil Boap Framework

4.1.2 Interface de l'outil Boap Framework

Comme nous l'avons précisé précedement, nous avons implémenté le module de calcul des métriques pour l'évaluation de la qualité des packages au sein de la plate-forme d'extraction des métriques implémentées par Alikacem et Sahraoui. En effet, la plate-forme contient un outil qui permet de définir des métriques suivant le méta-modèle correspondant au programme OO à analyser. La *figure4.1.1* présente un aperçu sur l'interface principale de l'outil, où on définit la métrique de cohésion, comme exemple.

D'après la *figure1.1.2* on remarque que l'outil est très semblable à n'importe quel éditeur de langage de programmation. En effet, l'explorateur dans la partie gauche contient le projet à analyser. Dans la figure, il s'agit du projet CPL. Les sous-dossiers du projet représentent l'ensemble des métriques définies suivant le langage PatOIS. La partie droite de la figure représente l'éditeur où l'utilisateur de la plate-forme peut définir ces métriques suivant le méta-modèle, à travers le langage PatOIS.

La partie infèrieure joue le rôle de la console où on affiche les résultats des calculs des mé-

triques ou des éventuelles erreurs de compilation. Dans le menu, nous avons deux volets : le volet *project*, contient les options de manipulation du projet [2]. Le deuxième volet *analysis,* permet de créer un nouveau fichier pour définir des métriques ou pour appliquer des modifications sur l'ensemble des métriques primitives implémentées. De plus, il contient le bouton de compilation (*check*) et le bouton d'exécution (*execute*).

4.2 Le langage PatOIS

4.2.1 Description du langage

La plate-forme d'extraction des métriques, *Boap Framework* permet d'évaluer la qualité d'un programme OO en se basant sur le méta-modèle établi à partir du code source. En revanche, la définition des métriques nécessite un mécanisme qui permet de récupérer les données du modèle et d'effectuer des opérations sur ces données[AS09]. À cet effet, le langage PatOIS [3] est mis en place pour permettre à l'utilisateur de la plate-forme de mieux exploiter le modèle, indépendamment du code source.

4.2.2 Les mécanismes du langage

L'accès aux données stockées dans l'instance du méta-modèle créé, nécessite des mécanismes qui permettent de définir les métriques d'une manière flexible et efficace. Le langage PatOIS est exprimé suivant quatre mécanismes de base :

1. La sélection : pendant le processus de transformation du code source en méta-modèle, plusieurs occurrences du programme sont créées suivant les différents concepts du paradigme orienté-objet. Dans l'instance du méta-modèle, les éléments du programme ont des relations entre eux. Plusieurs métriques OO nécessitent la sélection des instances d'un concept en particulier. Par exemple, le nombre d'interfaces dans un programme.
2. Le filtrage : la sélection permet d'extraire tous les éléments du même concept. En revanche, dans certains cas, la définition des métriques nécessite la sélection des éléments qui partagent une propriété particulière. Par exemple, le nombre des classes abstraites.

2. Sauvegarde, ouverture, fermeture,...
3. Primitives, Opérations, Iterator et acceSsor

3. La navigation : certaines métriques nécessitent la réunion de plusieurs concepts en même temps. En effet, la navigation permet d'explorer un élément et son voisinage à travers ses relations. Par exemple, la navigation dans l'ensemble des paramètres d'entrée d'une méthode.
4. Les opérations : ce mécanisme permet de transformer les ensembles récupérés à partir des trois premiers mécanismes, en valeurs numériques. Le langage supporte aussi les opérations arithmétiques et les comparateurs.

Les quatre mécanismes décrits précédemment rendent le langage PatOIS très expressif par rapport aux autres langages de description de métriques. En effet, ils permettent à l'utilisateur du langage de concevoir ses propres métriques d'une manière flexible et efficace.

4.2.3 Les caractéristiques du langage PatOIS

En se référant aux quatre mécanismes décrits dans la section précédente, le langage PatOIS a été conçu dans le but de faciliter l'extraction et le calcul des métriques à partir du méta-modèle. Les principales caractéristiques de ce langage sont les primitives, les opérations, les itérateurs et les accesseurs.

– Les primitives :

C'est la caractéristique de base de ce langage. En effet, les primitives sont codées dans la plate-forme et elles sont utilisées comme des fonctions de bibliothèques. Elles permettent de retourner un ensemble d'instances du méta-modèle pour un concept en particulier. Par exemple, *modules()*, où on retrouve l'ensemble des packages du programme.

Le deuxième type de primitives retourne un ensemble d'éléments reliés à un élément spécifique à travers une relation particulière. Par exemple, *int_dep (m)*, *ext_dep (c)*, ces deux métriques primitives retournent respectivement l'ensemble de dépendances internes d'un *module m* et le nombre de dépendances externes d'une *classe c*.

Le troisième type de primitives est celui que l'utilisateur définit lui même. Elles sont implémentées pour effectuer les procédures de calcul complexes que le langage ne peut pas exprimer.

– Les opérations :

La deuxième caractéristique du langage traite des opérations. En effet, le langage PatOIS supporte trois types d'opérations. Les opérations arithmétiques (sum, minus, max, etc.), la compa-

raison entre les valeurs numériques et entre les chaines de caractères (*String*). De plus, il y a les opérations sur les ensembles, tels que l'union, l'intersection, etc.

– Les accesseurs :

Ils permettent d'accéder à la propriété d'un élément à partir de l'instance du méta-modèle. Par exemple, *m.visibility* retourne la visibilité de la méthode m.

– Les itérateurs :

C'est la caractéristique la plus puissante du langage. En effet, un itérateur permet de manipuler les éléments d'un ensemble retourné. Leurs syntaxes sont simples, mais efficaces.

forAll (x : inputSet ; condition_clause ; SET AssignOperator expression) ;

Pour tout élément de l'ensemble *inputSet*, si la condition est vérifiée, la variable prédéfinie reçoit la valeur d'*expression*, sous la forme d'un ensemble, où nous pouvons ajouter les opérateurs de la cardinalité si on veut calculer la valeur numérique de l'ensemble retourné.

Le langage PatOIS est un langage qui a une syntaxe très simple. En effet, ce langage est facile à apprendre par les utilisateurs de la plate-forme. De plus, ce langage permet d'implémenter la majorité des métriques OO utilisées, tels que, la cohésion, le couplage, la dépendance des packages ou des classes.

4.2.4 Les types des métriques

Le langage PatOIS supporte trois types de métriques, les métriques basiques, les métriques complexes et les métriques basées sur d'autres métriques.

– Les métriques basiques : ce sont des métriques dont l'implémentation nécessite les opérateurs de cardinalités. Ces métriques se servent des métriques primitives implémentées au sein du langage, dans le but de construire de nouvelles métriques. D'après l'exemple suivant, on a ajouté l'opérateur de cardinalité sur la métrique primitive *classes,* pour obtenir une nouvelle métriques basique dite NC[4] qui mesure le nombre des classes dans un module *m*.

NC(m : module) : |classes (m)| ;

4. Number of Class

- Les métriques complexes : ce type de métriques s'appuient principalement sur le mécanisme d'itération. Par exemple, pour calculer le nombre de dépendances totales d'un module *TotaleModuleDep*, il faut mesurer les dépendances totales de chaque classe appartenant à ce module.

$$TotaleModuleDep(m : module) :$$
$$forAll(c : classes(m) ; ; SET+=|total\_dep(c)|) ;$$

- Les métriques basées sur d'autres métriques : le langage PatOIS permet également la définition d'une métrique en utilisant d'autres métriques prédéfinies. Ainsi, ces métriques seront considérées comme bibliothèque de fonctions. D'après l'exemple suivant, la métrique *NEntity* mesure le nombre d'entités dans un module *m*. Cette métrique a une struture complexe. Par conséquent, nous avons implémenté deux autres métriques *NC* et *NI*, qui mesurent respectivement le nombre des classes et le nombre d'interfaces dans *m*.

$$NC(m : module) : |classes\ (m)| ;$$
$$NI\ (m : module) : |interfaces(m)| ;$$
$$NEntity\ (m : module) : sum(set(NI(m)\ ,\ NC(m))) ;$$

4.3 Le module évaluateur

Le module évaluateur joue deux rôles importants dans la plate-forme de calcul et d'extraction des métriques. Il effectue les calculs des métriques décrites suivant le langage PatOIS et il joue le rôle d'interpréteur.

4.3.1 Calcul des métriques

Le module évaluateur prend en charge le fichier qui contient la description des métriques et il les évalue. En effet, le fichier est constitué de deux parties. La première partie contient la description des métriques. La deuxième contient les instructions opérationnelles, où l'utilisateur peut spécifier le format de sortie pour les résultats des métriques (table, histogramme ou texte).

La *figure4.3.1* est un exemple concret d'un fichier de description des métriques en langage PatOIS. On remarque deux parties distinctes du fichier. La première partie contient la description

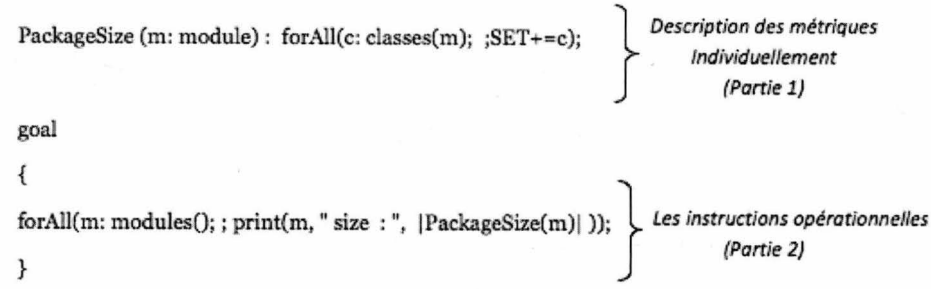


FIGURE 4.3.1 – Exemple d'un fichier de description des métriques en langage PatOIS

de la métrique *PackageSize*. Cette métrique prend en paramètre une variable de type *module* et elle définit la taille d'un package en fonction de l'ensemble de ces classes. Le type *module* fait référence à l'élément du méta-modèle sur lequel la métrique s'applique (voir *figure3.3.1*). En revanche, l'attribution d'un type à la métrique qu'on définit n'est nécessaire que lorsque la métrique vise un concept particulier dans l'instance du méta-modèle.

La description de la métrique est interprétée comme suit : pour toute classe c appartenant au module m, l'ensemble *SET* reçoit c. Lors de l'exécution, la métrique *PackageSize* recevra l'ensemble de toutes les instances des classes appartenant à m.

La deuxième partie du fichier est définie juste après le mot *goal*. Elle contient les instructions opérationnelles que l'utilisateur voudrait appliquer sur les métriques définies dans la partie précédente. D'après l'exemple de la *figure4.3.1*, la partie 2 est interprétée comme suit : pour l'ensemble de tous les modules m du méta-modèle, on affiche le module m suivi de la chaine de caractère "*size*", suivi de la métrique que nous avons définie. Les deux côtés de la cardinalité transforment la métrique *PackageSize* qui est sous la forme d'un ensemble en une valeur numérique qui désigne le nombre d'instances dans l'ensemble.

4.3.2 Interprétation du langage

Le module évaluateur joue le rôle d'un interpréteur du langage PatOIS. En effet, il parse le fichier de description des métriques et génère un graphe de syntaxe. Le langage a été implémenté suivant l'outil Flex Cup [5]. La phase de parsing est suivie d'une vérification de typage suivant les

5. Flex :flex.sourceforge.net
Cup : www2.cs.tum.edu/projects/cup/

primitives, les propriétés d'accès, etc.[AS09] Finalement, les noeuds de graphe créés pendant le parsing sont évalués suivant l'implémentation des différentes caractéristiques utilisées dans le script de description des métriques.

4.4 Les métriques implémentées

Le principal but de notre travail, c'est d'intégrer un module de calcul des métriques dans la plate-forme Boap Framework pour l'évaluation de la qualité des packages dans les programmes OO. Pour cela, nous avons implémenté des métriques appelées primitives dans le langage PatOIS. Ces primitives jouent le rôle d'une bibliothèque des métriques, mise à la disposions des utilisateurs qui voudront analyser la qualité de la modularisation d'un programme OO.

La définition des métriques primitives est basée sur notre perception de la dépendance inter et intra-modulaires dans les programmes OO. Une analyse détaillée de notre perception par rapport aux métriques des packages est présentée dans le chapitre 2. Dans ce qui suit, nous allons présenter l'ensemble des métriques primitives que nous avons implémentées. En effet, ces métriques seront les fonctions de bases que les utilisateurs pourront utiliser pour l'analyse de la qualité des packages.

De plus, nous allons présenter les quatre métriques principales de la conception des packages, la cohésion, le couplage, la stabilité et l'abstraction. Ces métriques sont définies suivant le langage PatOIS en fonction des métriques primitives que nous avons implémentées.

4.4.1 Les métriques primitives

Dans les programmes OO, la mesure de la dépendance entre les packages, passe par les classes. En effet, la nature abstraite des packages pose souvent des problèmes de précision par rapport à leurs métriques. Par conséquent, il est tout à fait légitime de mesurer les dépendances entre les classes pour pouvoir mesurer celle des packages.

L'implémentation des métriques primitives dans notre module de calcul est faite suivant les deux niveaux d'abstractions, les classes et les packages.

4.4.1.1 Les métriques primitives niveau classe

Nom : le nombre de dépendances sortantes

Acronyme : Outgoing_dep

Niveau : classe

Définition : le nombre de dépendances sortantes d'une classe C_i, est le nombre de dépendances de C_i vers le reste des classes du système OO.

Formule :

$$\text{Outgoing\_dep}\ (C_i) = \sum_{j=0}^{j=NC} |C_i \rightarrow C_j| \ / \ \{C_i \neq C_j\},$$

NC = nombre total des classes.

Le *Outgoing_dep*(C_i), est la somme de dépendances de C_i vers C_j avec j qui varie suivant le nombre total de classes du système et à condition que C_i et C_j soient deux classes différentes.

Nom : le nombre de dépendances entrantes

Acronyme : Incoming_dep

Niveau : classe

Définition : le nombre de dépendances entrantes d'une classe C_i, est le nombre de dépendances de l'ensemble des classes du système OO vers la classe C_i.

Formule :

$$\text{Incoming\_dep}\ (C_i) = \sum_{j=0}^{j=NC} |C_i \leftarrow C_j| \ / \ \{C_i \neq C_j\},$$

NC = nombre total des classes.

Le *Incoming_dep* (C_i), est le nombre de dépendances de C_j vers C_i avec j qui varie suivant le nombre de classes du système et à condition que C_i et C_j soient deux classes différentes.

Nom : le nombre total de dépendances

Acronyme : total_dep

Niveau : classe

Définition : le nombre total de dépendances d'une classe C_i, est le nombre de dépendances entrantes et sortantes de cette classe avec l'ensemble des classes du système OO.

Formule :

$$\text{total\_dep}\ (C_i) = \sum_{j=0}^{j=NC} |C_i \leftarrow C_j| \bigcup |C_i \rightarrow C_j| \ / \ \{C_i \neq C_j\},$$

NC = nombre total des classes dans le système OO

Le *total_dep* (C_i), est l'union des deux sous-ensembles, les dépendances sortantes ($C_i \rightarrow C_j$) et les dépendances entrantes ($C_i \leftarrow C_j$), avec j qui varie suivant le nombre de classes du système et à condition que C_i et C_j soient deux classes différentes.

Nom : le nombre de dépendances externes

Acronyme : ext_dep

Niveau : classe

Définition : le nombre de dépendances externes d'une classe C, est la somme des dépendances sortantes et entrantes externes de la classe C avec le reste des classes du système OO, appartenant à un package différent de celui de la classe C.

Formule :

$$\text{ext\_dep}\ (C_i) = \sum_{j=0}^{j=NC} |C_i \rightarrow C_j| + |C_i \leftarrow C_j| \ / \ \{C_i \in P, C_j \in P_k, P \neq P_k\}, k \sim \{0 \ldots NP\}$$

NC = nombre total des classes dans le système.

NP = nombre tatal des packages dans le système.

Le *ext_dep* (C_i), est la somme de dépendances de C_i vers C_j et de C_j vers C_i, à condition que les classes qui sont en dépendance avec C_i appartiennent à un package différent.

Nom : le nombre de dépendances internes

Acronyme : int_dep

Niveau : classe

Définition : le nombre de dépendances internes d'une classe *C,* est la somme de dépendances entrantes et sortantes internes de la classe avec le reste des classes appartenant au même package.

Formule :

$$\text{int\_dep}\ (C_i) = \sum_{j=0}^{j=NC(P)} |C_i \rightarrow C_j| + |C_i \leftarrow C_j| \ / \ \{C_i \in P, C_j \in P, C_i \neq C_j\}$$

NC(P) = nombre des classes dans le package p.

Le int_dep(C_i), est la somme des dépendances de C_i vers C_j et de C_j vers C_i avec C_i et C_j deux classes différentes appartenant au même package.

Nom : le nombre de dépendances sortantes externes

Acronyme : Outgoing_ext_dep

Niveau : classe

Définition : le nombre de dépendances sortantes externes d'une classe *C*, c'est la somme des dépendance sortantes externes de la classe C avec le reste des classes du système OO appartenant à un package différent de celui de la classe *C.*

Formule :

$$\text{Outgoing\_ext\_dep}\ (C_i) = \sum_{j=0}^{j=NC} |C_i \rightarrow C_j| \ / \ \{C_i \in P, C_j \in P_k, P \neq P_k\}, k \sim \{0 \ldots NP\}$$

NC = nombre total des classes dans le système.

NP = nombre tatale des packages dans le système.

Le *Outgoing_ext_dep* (C_i), est la somme de dépendances de C_i vers C_j, à condition que les classes avec lesquelles C_i est en dépendance, et qui appartiennent à un package différent.

Nom : le nombre de dépendances entrantes externes

Acronyme : Incoming_ext_dep

Niveau : classe

Définition : Le nombre de dépendances entrantes externes d'une classe C, est la somme des dépendances entrantes externes de la classe C avec le reste des classes du système OO appartenant à un package différent de celui de la classe C.

Formule :

$$\text{Incoming\_ext\_dep}\ (C_i) = \sum_{j=0}^{j=NC} |C_i \leftarrow C_j| \ / \ \{C_i \neq C_j, C_i \in P, C_j \in P_k, P \neq P_k\},$$
$$k \sim \{0 \ldots NP\}$$

NC = nombre total des classes dans le système.

NP = nombre tatal des packages dans le système.

Le Incoming_ext_dep (C_i), est la somme de dépendances de C_j vers C_j, à condition que les classes qui sont en dépendance avec C_i appartiennent à un package différent.

4.4.1.2 Les métriques primitives niveau package

L'intégration d'un module de calcul des métriques pour l'analyse de la qualité des packages nécessite entre autre l'implémentation d'un ensemble des métriques primitives au niveau des packages. En effet, ces métriques se basent sur les primitives que nous avons définies précédemment au niveau des classes. Ceci est tout à fait normal puisque la dépendance entre deux classes implique systématiquement une dépendance entre leurs packages respectifs. En recanche, nous avons repris la définition de quelques métriques de packages, tels que le couplage afférent et le couplage efférent.

Nom : le nombre total de dépendances

Acronyme : total_dep

Niveau : package

Définition : le nombre total de dépendances d'un package P, est la somme des nombres de dépendances totales de toutes les classes de ce package.

Formule :

$$\text{total\_dep}\ (P) = \sum_{i=0}^{i=NC(P)} total\_dep(C_i)$$

NC(P) = nombre des classes dans le package p.

Le total_dep d'un package P, est la somme de dépendances totales de l'ensemble des classes de P, identifiées par C_i, avec i qui varie de 0 à $NC(P)$, le nombre des classes du package P.

Nom : le nombre de dépendances externes

Acronyme : ext_dep

Niveau : package

Définition : le nombre de dépendances externes d'un package P, est la somme du nombre de dépendances externes de toutes les classes appartenant à ce package.

Formule :

$$\text{ext\_dep}\ (P) = \sum_{i=0}^{i=NC(P)} ext\_dep(C_i)$$

NC(P) = nombre de classes dans le package p.

Le *ext_dep* d'un package P, est la somme de dépendances externes de l'ensemble des classes de P identifiées par C_i, avec i qui varie de 0 à $NC(P)$, le nombre des classes du package P.

Nom : le nombre de dépendances internes

Acronyme : int_dep

Niveau : package

Définition : le nombre de dépendances internes d'un package P, est la somme du nombre de dépendances internes de toutes les classes appartenant à ce package.

Formule :

$$\text{int\_dep}\ (P) = \sum_{i=0}^{i=NC(P)} int\_dep(C_i)$$

NC(P) = nombre de classes dans le package p.

Le *int_dep* d'un package P, est la somme des dépendances internes de l'ensemble des classes de P identifiées par C_i, avec i qui varie de 0 à $NC(P)$, le nombre de classes dans le package P.

Nom : le nombre de dépendances sortantes

Acronyme : Outgoing_dep

Niveau : package

Définition : le nombre de dépendances sortantes d'un package *P*, est la somme du nombre de dépendances sortantes externes de toutes les classes appartenant à ce package.

Formule :

$$\text{Outgoing\_dep}\ (P) = \sum_{i=0}^{i=NC(P)} Outgoing\_ext\_dep(C_i)$$

NC(P) = nombre des classes dans le package p.

Le *Outgoing_dep* d'un package *P*, est la somme de dépendances sortantes externes de l'ensemble des classes de *P* identifiées par C_i, avec *i* qui varie de 0 à $NC(P)$, le nombre des classes dans le package *P*.

Nom : le nombre de dépendances entrantes

Acronyme : Incoming_dep

Niveau : package

Définition : le nombre de dépendances entrantes d'un package *P*, est la somme du nombre de dépendances entrantes externes de toutes les classes appartenant à ce package.

Formule :

$$\text{Incoming\_dep}\ (P) = \sum_{i=0}^{i=NC(P)} Incoming\_ext\_dep(C_i)$$

NC(P) = nombre de classes dans le package p.

Le *Outgoing_dep* d'un package *P*, est la somme des dépendances entrantes externes de l'ensemble des classes de *P* identifiées par C_i, avec *i* qui varie de 0 à $NC(P)$, le nombre des classes dans le package *P*.

Nom : le couplage afférent

Acronyme : Ca

Niveau : package

Définition : le couplage afférent d'un package P, est le nombe de classes qui dépendent d'une ou plusieurs classes appartenant au package P [Mar94].

Formule :

$$\mathrm{Ca}(P) = \sum_{i=0}^{i=NC(P)} |C_j| \,/\, \{C_i \leftarrow C_j, C_i \neq C_j, C_i \in P, C_j \in P_k, P \neq P_k\}, k \sim \{0 \ldots NP\}$$

NC(P) = nombre de classes dans le package p.

NP = nombre tatal de packages dans le système.

Le couplage afférent d'un package P, est le nombre de classes C_j qui sont en dépendance avec C_i [6]. L'ensemble des dépendances $\{C_i \leftarrow C_j\}$ doit vérifier les conditions suivantes : C_i et C_j deux classes différentes. L'ensemble des classes C_j doit appartenir à un package différent de celui de C_i. De plus, les dépendances redondantes dans l'ensemble ne sont pas comptabilisées.

Nom : le couplage efférent

Acronyme : Ce

Niveau : package

Définition : le couplage efférent d'un package P, est le nombe de classes appartenant à P et qui ont une dépendance avec des classes à l'extérieur du package P [Mar94].

Formule :

$$\mathrm{Ce}(P) = \sum_{i=0}^{i=NC(P)} |C_j| \,/\, \{C_i \rightarrow C_j, C_i \neq C_j, C_i \in P, C_j \in P_k, P \neq P_k\}, k \sim \{0 \ldots NP\}$$

NC(P) = nombre de classes dans le package p.

NP = nombre tatal de packages dans le système.

Le couplage afférent d'un package P, est le nombre de classes C_j qui sont en dépendance avec C_i [7]. L'ensemble des dépendances $\{C_i \rightarrow C_j\}$ doit vérifier les conditions suivantes : C_i et C_j deux classes différentes. L'ensemble des classes C_j doit appartenir à un package différent de celui de C_i. De plus, les dépendances redondantes dans l'ensemble ne sont pas comptabilisées.

---

6. l'ensemble des classes de P identifiées par C_i
7. l'ensemble des classes de P identifiées par C_i

4.4.2 Les métriques de packages implémentées dans le langage PatOIS

D'après notre étude approfondie de la littérature sur les métriques de conception, définies dans le but d'évaluer la qualité des packages dans les systèmes OO, un consensus s'est formé autour de quatre métriques : la cohésion, le couplage, l'instabilité et l'abstraction. Ces métriques ont été proposées principalement par R.Martin [Mar94].

En nous basant sur les métriques primitives que nous avons implémentées dans le module de calcul des métriques, nous avons aussi implémenté les quatre métriques de packages au niveau de l'outil Boap Framework, en utilisant le langage *PatOIS*. Ceci est dans le but d'exploiter l'efficacité des métriques primitives dans l'implémentation des métriques de packages de haut niveau, via le nouveau langage.

4.4.2.1 La cohésion

La métrique de cohésion permet d'évaluer un package par rapport à ses dépendances internes. En effet, un package de bonne qualité est celui qui a un niveau de cohésion maximale. La cohésion d'un package est mesurée à travers les dépendances internes dites intra-modulaires et par rapport aussi aux dépendances totales du package [ADSA09].

Formule :

$$Cohesion\,(P) = \frac{int\_dep(p)}{Total\_dep(p)}$$

La *figure4.4.1* représente l'implémentation de la métrique de cohésion au niveau du langage *PatOIS*. D'après la formule, la cohésion est composée de deux autres métriques, *IntraModule-Dependance* et *TotalModuleDependance* qui s'appliquent sur le concept associé aux packages dans l'instance du méta-modèle (les modules). La mesure de cette métrique varie dans [0,1]. Lorsque la cohesion est égale à 1, cela implique que le package est cohésif à son maximum. En revanche, lorsque la cohésion est égale à 0, cela signifie que le package est très mauvais du point de vue cohésion.

IntraModuleDependance : mesure le nombre de dépendances internes d'un module m en fonction de l'ensemble de ses classes classes(*m*). Cette métrique est une métrique complexe dans laquelle on mesure le *int_dep* de chaque classe appartenant au module *m*. Finalement, on retourne la cardinalité de l'ensemble trouvé.

TotalModuleDependance : mesure le nombre total de dépendances d'un module m. On mesure les dépendances totales de chaque classe et on retourne la cardinalité de l'ensemble.

Cohésion : cette métrique prend en paramètre un module *m*, et mesure le ratio entre les deux métriques, IntraModuleDependance et TotalModuleDependance.

```
IntraModuleDependance (m:module) : forAll (c:classes(m); ;SET+=|int_dep(c)|);
TotalModuleDependance(m:module): forAll(c:classes(m); ;SET+=|total_dep(c)|);

Cohesion(m:module): IntraModuleDependance(m) / TotalModuleDependance(m);

//-------------------------------------------------------------------------
goal {
      forAll(m: modules(); ; print(m, " Cohesion est : ", Cohesion(m)));
}
```

FIGURE 4.4.1 – Implémentation de la métrique de cohésion en langage PatOIS

4.4.2.2 Le couplage

La métrique de couplage permet d'évaluer un package par rapport à ses dépendances sortantes et entrantes externes avec les autres packages du système. R.Martin identifie ces dépendances par le couplage efférent et afférent [Mar05]. La métrique de couplage est calculée en fonction des deux métriques primitives, *Ca* et *Ce*.

Formule :

$$Couplage\,(P) = 1\text{-}\,\frac{Ce(p)\cup Ca(p)}{Total\_dep(p)}$$

La métrique de couplage au niveau des packages a été souvent le sujet de plusieurs recherches, à cause des conflits qui tournent autour de sa définition. Dans l'implémentation de cette métrique au niveau du langage *PatOIS*, on s'est inspiré de la formule présentée par H.Abdeen et al.[ADSA09]. En effet, ils considèrent que la métrique de couplage doit prendre en compte les dépendances externes entrantes et sortantes d'un package, à savoir, les deux métriques primitives *Ce* et *Ca*, que nous avons définies dans la section précédente.

La *figure4.4.2* représente notre implémentation de la métrique de couplage des packages au niveau du langage *PatOIS*. Cette métrique est composée de deux autres métriques *CouplageAffrent* et *CouplageEffrent*. Ces deux métriques sont définies en fonction des primitives *Ce* et *Ca* appliquées sur le concept module, au niveau de l'instance du méta-modèle. La troisième métrique *CouplageUnion* contient l'unification des deux ensembles retournés par les deux premières métriques. Enfin, on applique la formule de couplage au niveau de la dernière métrique : *Couplage (m)*.

De même que la cohésion, la mesure de la métrique de couplage varie dans [0,1]. En effet, lorsque la valeur du couplage est égale à 1, cela implique que le package est de très mauvaise qualité d'un point de vue couplage. En revanche, la valeur 0 indique que le package n'a pas de couplage externe avec d'autres packages dans le système OO.

```
CouplageAffrent(m:module): |Ca(m)|;

CouplageEffrent(m:module): |Ce(m)|;

CouplageUnion(m:module):CouplageEffrent(m) + CouplageAffrent(m);

TotalModuleDependance(m:module): forAll(c:classes(m); ;SET+=|total_dep(c)|);

Couplage(m:module): 1- (CouplageUnion(m) /  TotalModuleDependance(m)

//-------------------------------------------------------------------------
goal
    {

     forAll(m: modules(); ; print(m, " Couplage(Q) : ", Couplage (m)));


    }
```

FIGURE 4.4.2 – Implémentation de la métrique de couplage en langage PatOIS

4.4.2.3 L'instabilité

La métrique de l'instabilité permet d'evaluer la volatilité d'un package dans un environnement OO. Un package est dit instable à cause de sa sensibilité face aux éventuelles modifications dans le modèle OO associé au programme en cours d'analyse. En effet, si un simple changement au

niveau des dépendances dans le modèle entraine des changements en cascade sur l'ensembe des classes du package, par conséquent, ce package est instable.

Formule :

$$I(P) = \frac{Ce(p)}{Ce(p)+Ca(p)}$$

La *figure4.4.3* représente l'implémentation de la métrique de l'instabilité au niveau du langage *PatOIS*. Cette métrique est mesurée à partir de deux autres métriques, le couplage efférent et le couplage afférent. Premièrement, on a la métrique *CouplageEffrent*, elle mesure le nombre de couplage efférent d'un package m en utilisant la primitive *Ce* appliquée au niveau de l'ensemble des classes du package. De même, la deuxième métrique *CouplageAffrent* mesure le couplage afférent du package. Finalement, on applique la formule au niveau de la troisième métrique pour mesurer l'instabilité du package.

La mesure de cette métrique varie dans l'intervalle [0,1], où le 0 indique que le package est stable au maximum. Cependant, lorsque $I = 1$, cela signifie que le package est instable au maximum.

```
CouplageAffrent(m:module): |Ca(m)|;

CouplageEffrent(m:module): |Ce(m)|;

Instability (m:module):

       CouplageEffrent(m) / (CouplageAffrent(m) + CouplageEffrent(m)) ;

//-------------------------------------------------------------------------
goal

{

forAll(m: modules(); (CouplageAffrent(m) + CouplageEffrent(m)) != 0;
                    print(m, " Instability :  ",  Instability(m) ));

}
```

FIGURE 4.4.3 – Implémentation de la métrique Instabilité en langage PatOIS

4.4.2.4 L'abstraction

La métrique d'abstraction d'un package permet de mesurer la flexibilité du package face aux éventuelles extensions possibles d'un modèle OO. En effet, cette métrique est complémentaire à la métrique de l'instabilité. Un package qui a une stabilité maximale ($I = 0$) cela signifie que ce package est complètement rigide au niveau du modèle. Or, ceci peut être très nuisible dans le cas où l'on veut étendre le modèle OO associé au programme que nous voulons analyser. Par conséquent, la métrique d'abstraction a été définie dans le but d'évaluer la qualité d'un package en terme d'extensibilité.

Formule :

$$A(P) = \frac{NCA(P)}{NC(P)}$$

NCA(P) = nombre des classes abstraites dans le package p.

NC(P) = nombre tatal des classes dans le package p.

On présente dans la *figure4.4.4* l'implémentation de la métrique d'abstraction au niveau de l'éditeur du langage *PatOIS*. L'abstraction est une métrique composée de deux autres métriques. Premièrement, on définit la métrique NC [8] ; c'est une métrique basique, mesurée à partir de la primitive *classes(m)*, qui retourne l'ensemble des classes d'un module m. Deuxièmement, on définit la métrique *NCA* [9] ; c'est une métrique de type composée. En effet, *NCA* est mesurée en fonction de l'iterateur *forAll*, qui itère sur l'ensemble des classes de m et vérifie que la classe est abstraite. Si c'est le cas, on ajoute la classe c à l'ensemble retourné. Finalement, on applique la formule adoptée pour le calcul de l'abstration. C'est le ratio entre *NCA(m)* et *NC(m)*.

La mesure de la métrique d'abstraction varie dans [0,1]. Si $A(P) = 1$, cela implique que le package est completement abstrait. En revanche, si $A(p) = 0$, cela signifie que le package P ne contient aucune classe abstraite.

8. Nombre total des classes dans un module m
9. Nombre des classes abstraites dans un module m

```
NC(m: module) : |classes (m)|;

NCA(m:module) : |forAll (c:classes(m); isAbstract(c) == true ; SET += c)|;

Abstraction (m:module): NCA (m) / NC (m);
//-------------------------------------------------------------------
 goal {
    forAll(m: modules(); ; print(m, " Abstraction: ",  Abstraction(m)));
}
```

FIGURE 4.4.4 – Implémentation de la métrique d'abstraction en langage PatOIS

Conclusion

Melton H. et Tempero E. [MT07] estiment que la qualité des packages dans un système OO, d'un point de vue conceptuel, a un effet direct sur la réutilisabilité et la testabilité des différents modules d'un système OO. L'analyse de la qualité des packages est un attribut majeur pour l'évaluation de la qualité des systèmes OO. Dans la littérature, plusieurs métriques de package ont été proposées, mais l'efficacité de ces métriques était toujours un sujet de critique par plusieurs chercheurs dans le domaine, à cause de la nature abstraite des packages dans le paradigme de l'orienté-objet.

La plate-forme de calcul et d'extraction des métriques, développée par Alikacem El. et Sahraoui ne supporte pas les métriques de packages. À cet effet, nous avons implémenté un module de calcul des métriques dans le but d'étendre la plate-forme, pour permettre aux futurs utilisateurs de mesurer et d'analyser la qualité de la modularisation des programmes OO. Tel que cela a été expliqué dans la première section de ce chapitre, la plate-forme fonctionne suivant deux types de métriques. Tout d'abord, on a les métriques primitives qui sont implémentées directement au sein de la plate-forme et qui sont utilisées comme des fonctions de bibliothèque que l'utilisateur exploite pour la définition d'autres métriques, via le langage PatOIS.

Pour la réalisation de notre module de calcul des métriques, nous avons opté pour des métriques conceptuelles qui visent à mesurer la qualité des packages à partir d'un modèle OO.

En effet, nous avons implémenté au sein de ce module un ensemble de métriques primitives niveau classes et packages qui mesurent les différents types de dépendances reliées aux deux concepts, les *classes* et les *packages*. Par la suite, nous avons présenté l'implémentation des quatre métriques principales des packages, le *couplage*, la *cohésion*, *l'instabilité* et *l'abstraction*, au niveau du langage PatOIS, en utilisant les métriques primitives.

Dans le prochain chapitre, nous allons mettre en oeuvre notre module de calcul des métriques sur des projets concrets pour l'évaluation de la qualité de leurs packages. Ceci nous permettra de mieux tester l'efficacité de nos métriques sur des projets réels.

Chapitre 5

Étude de cas et résultats des métriques

Nous avons présenté dans le chapitre précédent, les différentes métriques de packages, que nous avons implémentées au sein de la plate-forme. Dans notre travail, il existe trois types de métriques, à savoir, les métriques primitives, les métriques complexes et les métriques basées sur d'autres métriques.

Dans le but d'évaluer l'ensemble de ces métriques, nous avons choisi une approche méthodologique. Notre étude portera sur trois cas, CPL, PADL et PayrusForTest. Les deux projets sont écrits en *Java*. De plus, ils sont différents d'un point de vue architecture et taille. Dans un premier lieu, nous allons définir les différents axes de notre étude, et établir les méta-modèles associés à chaque projet. Par la suite, nous allons décrire l'ensemble des métriques primitives implémentées comme bibliothèque des fonctions dans le langage PatOIS. Ensuite, nous allons analyser les résultats des métriques implémentées en langage PatOIS en fonction des métriques primitives.

Il est important que les résultats de ces métriques soient significatifs et efficaces. Les métriques n'ont pas d'importance si elles ne sont pas efficaces. Par conséquent, l'efficacité et l'intégrité des résultats sont deux critères très importants dont on doit tenir compte durant la phase d'expérimentation.

Pour considérer que ces métriques sont valides, il faut que les résultats reflètent la réalité. L'interprétation de ces métriques sera basée essentiellement sur notre perception de la qualité des packages, discutée dans le chapitre 2. Finalement, nous allons présenter le graphe de dépendances au niveau des packages et des entités, décrit suivant le langage DOT [1].

5.1 Définition du domaine de l'étude

- Objet de l'étude : la qualité de la modularisation dans les systèmes OO.
- Objectif de l'étude : évaluer l'efficacité des métriques implémentées.
- Perspective de l'étude : améliorer la qualité des systèmes OO d'un point de vue structurel.

1. http ://fr.wikipedia.org/wiki/DOT_(langage)

5.2 Génération des instances du méta-modèle

Les tests concernant le méta-modèle généré à partir du code source, sont inscrits dans des fichiers logs. Ces fichiers nous permettront de suivre la création des objets relatifs aux événements de dépendance, détectés durant la phase de *parsing* du code source. En effet, pour chaque élément créé durant la phase de mapping, une information est inscrite au niveau des fichiers log pour confirmer la construction de l'objet dans la mémoire.

La fi*gure 5.2.1*, est un aperçu des fichiers log durant le mapping des trois projets de tests, à savoir CPL, PADL et PapyrusForTest. Ces fichiers contiennent les traces de création des instances du méta-modèle pour chaque projet. En effet, les fichiers log servent à vérifier manuellement l'intégrité de l'instance du méta-modèle avec le code source.

D'après la conception générique du méta-modèle que nous avons adoptée dans la plate-forme *(figure 3.3.1)*, on remarque que l'élément *Entity* est en relation avec l'élément *Feature*. Ce dernier contient l'ensemble des attributs et méthodes relatives à chaque entité détectée durant le parsing. Dans la *figure 5.2.1*, on présente un aperçu de quelques éléments crées dans l'instance du méta-modèle, pour chaque programme à partir de son code source. Chaque élément *Feature* correspond à un attribut ou à une méthode, détecté lors de la visite de l'entité en question. Par exemple, l'entité *FieldInfo*, dans l'application CPL, a 19 *Features* listés juste en dessous.

5.3 Métriques implémentées

L'intégration d'un module de calcul et d'analyse de métriques pour les packages dans la plate-forme, necessite l'implémentation d'un ensemble de métriques primitives. Ces métriques jouent le rôle d'une bibliothèque de fonctions qui seront appelées lors de l'implémentation d'autres métriques de plus haut niveau, avec le langage PatOIS. La *table 5.3.1* décrit l'ensemble des métriques primitives implémentées, et leurs niveau d'implémentation par rapport à deux composantes du méta-modèle, à savoir, les packages et les classes.

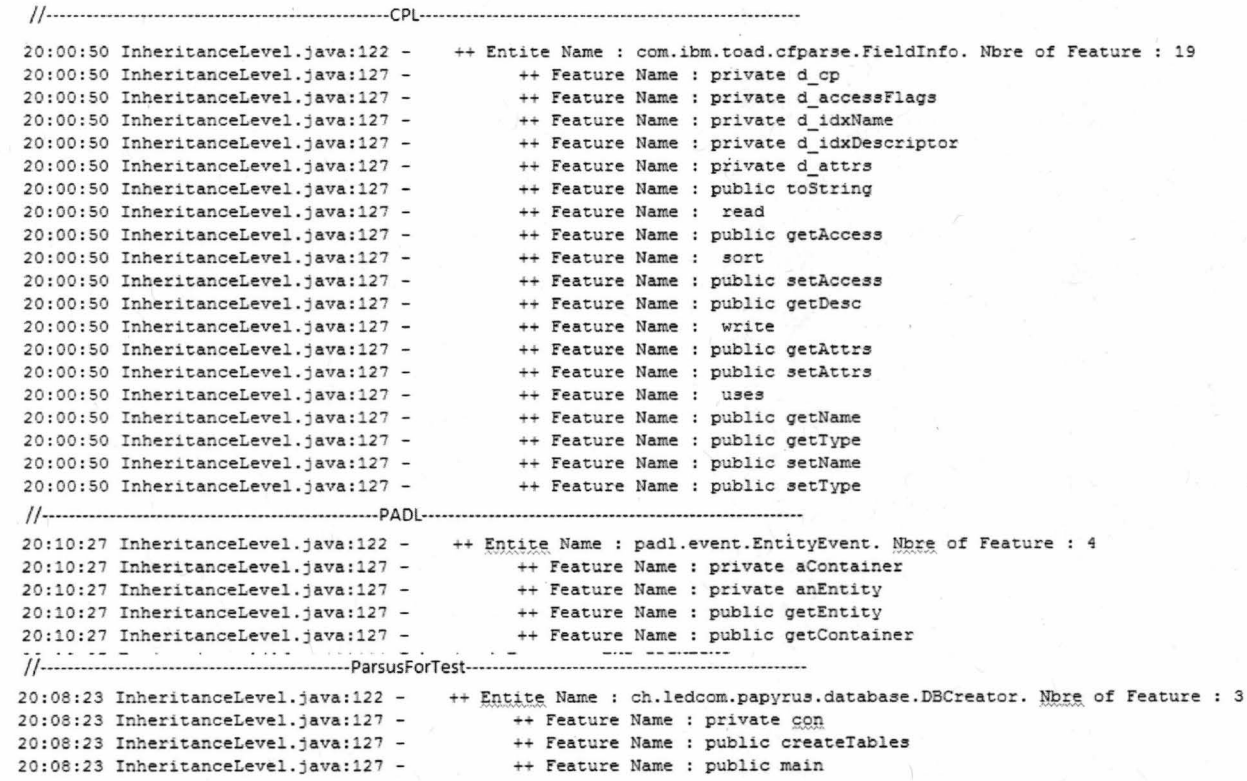


FIGURE 5.2.1 – Impression écran d'un fichier log cas : CPL, PADL et PapyrusForTest

OutDep	le nombre de dépendances sortantes avec le reste du système	Package, Classe
IncDep	le nombre de dépendances entrantes par rapport au reste du système	Package, Classe
TotalDep	le nombre total de dépendances avec tout le système	Package, Classe
ExtDep	le nombre de dépendances externes en termes de package	Package, Classe
IntDep	le nombre de dépendances internes en termes de package	Package, Classe
IntOutDep	le nombre de dépendances sortantes internes par rapport au package	Classe
IntIncDep	le nombre de dépendances entrantes internes par rapport au package	Classe
ExtOutDep	le nombre de dépendances sortantes externes par rapport au package	Classe
ExtIncDep	le nombre de dépendances entrantes externes par rapport au package	Classe
Ce	le nombre des classes qui ont des dépendances externes par rapport au package	Package
Ca	le nombre des classes externes qui ont des dépendances avec une ou plusieurs classe du package	Package

TABLE 5.3.1 – Les métriques au niveau des packages et des classes

5.4 Résultats des métriques et interprétation

5.4.1 Métriques complexes

Les métriques complexes constituent le deuxième type supporté par le langage PatOIS. En se basant sur les métriques primitives présentées dans la section précédente, nous avons implémenté un nouveau ensemble de métriques de plus haut niveau pour mesurer les dépendances des packages. La *table 5.4.1* contient les résultats de ces métriques appliquées sur les deux cas, CPL et PapyrusForTest. Nous avons choisi d'utiliser ces deux applications comme cas de test

pour leur simplicité et leurs modèles OO différents. Le but est aussi de tester l'efficacité des métriques dans deux environnements différents.

Étude de cas : CPL

L'application CPL est une application de *parsing* de code source, qui comporte 17 packages. Le nombre total d'entités $N(E)$ est 272. On remarque que la dispersion des classes dans les packages n'est pas équilibrer dans le sens où on trouve des packages qui ont un nombre de classes faible par rapport à d'autres packages dans le système CPL. Par exemple, Certains package contient moins que trois classes (*cfparse*, *attributes*, *util*, *awt*, *help*, etc). Cependant, d'autres packages ont un nombre de classes beaucoup plus important qui arrive jusqu'à 105 classes dans les packages *V14.node* et *V15.node*. Ceci, est considéré comme un mauvais signe par rapport à la structuration de packages dans cette application.

La métrique *TotalDep* nous a permis d'identifier les packages qui ont un flux de dépendances assez élevé par rapport aux autres packages du système, à savoir, *V14*, *V14.nodes*, *V14.visitors*, *V15*, *V15.nodes* et *V15.visitors*. Ainsi, ces packages deviennent prioritaires dans notre processus d'analyse. En effet, l'ensemble des packages qui ont un taux de dépendance total assez élevé, sont considérés comme les packages principaux de l'application, d'un point de vue conceptuel et fonctionnel. En effet, dans l'application CPL les packages *V14* et *V15* jouent le rôle d'une couche supérieure qui regroupe l'ensemble des patrons visiteurs utilisés par l'application. Ceci, met en évidence l'aspect de l'héritage simple dans le langage Java. Par conséquent, ces package sont fortement liées entre eux, ce qui est parfaitement on accord avec les résultats de cette métrique. D'autre part, cette métrique est essentielle pour le calcul du couplage et de la cohésion des packages.

Les métriques *InternalDep* et *ExternalDep* concernent respectivement les relations de dépendances intra et inter-modulaires. La comparaison des résultats de ces deux métriques, nous permet de comprendre la nature des dépendances du package. Par exemple, pour les packages *V14.nodes* et *V15.nodes*, le nombre de dépendances externes est beaucoup plus élevé que le nombre de dépendances pour le reste des packages. De plus, le nombre de dépendances externes pour ces deux packages, est plus important que le nombre de dépendances internes. Par conséquent, ces packages sont plus orientés vers l'externe que vers l'interne. Les classes de ces deux packages dépendent des classes appartenant aux packages *V14* et *V15* à cause de la

relation d'héritage, ce qui confirme que ces packages sont plutôt orientés vers l'externe. Mais, cela ne suffit pas pour donner une appréciation précise sur la qualité de ce package en terme de structuration.

V14.visitors et *V15.visitors* sont les deux packages qui ont le plus grand écart entre les résultats de ces deux métriques, ($InternalDep(V14.visitors) = 4$, $ExternalDep(V14.visitors) = 1180$) et ($InternalDep(V15.visitors) = 4$, $ExternalDep(V15.visitors) = 1490$). En effet, les dépendances externes dépassent de loin les dépendances internes. Cela signifie que le package aura un mauvais taux de couplage et de cohésion. D'ailleurs, ces deux métriques jouent un rôle important aussi dans l'implémentation du couplage et de la cohésion.

Les deux métriques *OutDep* et *IncDep* dérivent de la métriques *ExternalDep*. Elles désignent le nombre de dépendances sortantes et entrantes d'un package. Les résultats de ces métriques permettent d'identifier d'une manière plus claire la nature de dépendances externes d'un package. Ces deux métriques sont très significatives par rapport à l'instabilité des packages. De plus, elles sont très utiles pour la validation du principe SDP que nous avons présenté dans le premier chapitre. D'après la *table 5.4.1*, le package *V15.visitors* a le plus grand nombre de dépendances sortantes dans le système CPL ($OutDep = 1038$). Cependant, le package *V15.nodes* a le plus grand nombre de dépendances entrantes, parmis les autres packages du système ($IncDep = 2232$). Ainsi, nous pouvons conclure que ce package est un point sensible dans un éventuel processus de maintenance, vu qu'il contient des services très convoités par le reste des packages. Les métriques *Ce* et *Ca* représentent respectivement les dépendances sortantes et entrantes des packages. Dans le cas CPL, les packages *V14.visitors* et *V15.visitors* ont les résultats les plus élevés par rapport au reste des packages (92, 114).

Les résultats des métriques présentées dans la *table 5.4.1* nous ont permis de comprendre certains points par rapport à l'architecture des packages dans l'application CPL. D'un point de vue fonctionnel, l'architecture des packages dans le modèle du cas CPL, est centralisée autour de quelques packages. De plus, l'absence d'un équilibre entre les dépendances externes et internes et les dépendances sortantes et entrantes, confirme que ces packages sont mal organisés et difficile à maintenir.

Étude de cas : PADL

L'application PADL est consitituée de 75 classes dispersées sur 5 packages de tailles différentes, et implémentées en *Java*. D'après la *table 5.4.1,* le package *kernel* a le plus grand nombre de dépendances totales ($TotalDep = 532$), Cependant les packages *event*, *exception* et *path* ont respectivement un nombre de dépendances totales de, 62, 53 et 37. On remarque que le *TotalDep* du package *kernel* dépasse le reste des packages. Ceci s'explique par le fait que le package kernel contient le plus grand nombre de classes par rapport au reste des packages du système PADL. Ainsi, on peut le considérer comme le package principal de l'application.

Dans ce cas, les résultats des métriques *InternalDep* et *ExternalDep* ont montré que les classes du package *exception* n'ont aucune relation de dépendances entre elles ($InternalDep = 0$ et $ExternalDep = 53$). Ainsi, on conclu que ce package est relié en totalité avec le reste des packages du système.

D'après les résultats des métriques *OutDep* et *IncDep,* on note que la totalité des dépendances externes des deux packages *analysis* et *path*, sont des dépendances sortantes ($OutDep(analysis) = 2, IncDep(analysis) = 0$) et ($OutDep(path) = 29, IncDep(path) = 0$). Ainsi, on remarque que ces deux packages sont les plus susceptibles d'être impactés dans un processus de modification. Cependant, la totalité des dépendances externes du package *exception* sont des dépendances entrantes ($OutDep(exception) = 0, IncDep(exception) = 53$).

Les résultats des métriques *Ce* et *Ca* permettent de mesurer le nombre de classes impliquées dans les dépendances entrantes ou sortantes d'un package. Par exemple, d'après les résultats de la *table 5.4.1*, on note que le package *expection* a un taux de couplage afférent ($Ca(exception) = 15$). Ce package est un package producteur par rapport au reste des packages du système PADL. De plus, il y a 15 classes appartenant à d'autres packages, qui sont impliquées dans des relations de dépendances avec les classes du package *exception*. D'autre part, le package *kernel* a un nombre de couplage efférent et afférent ($Ce(kernel) = 5$ et $Ca(kernel) = 8$). D'après ces résultats, on remarque que le nombre des classes externes au packages *kernel* et qui sont en relation avec les classes de ce package, ne sont pas nombreuses, malgré que ce package, a un grand nombre de dépendances externes ($ExternalDep(kernel) = 114$). Cela signifie que les classes de ce package sont fortement liées avec quelques classes externes appartenant à des packages différents.

Étude de cas : PapyrusForTest

L'application PapyrusForTest est une application *Java* simple, L'architecture des packages est conforme au modèle de développement MVC (Model View Control). Elle est constituée de 5 packages avec un nombre total d'entités, $N(E) = 18$.

D'après la *table 5.4.1,* les packages *Gui* et *Utils* ont le plus grand nombre de dépendances totales par rapport aux autres packages de l'application ($TotalDep(Gui) = 48, TotalDep(Utils) = 38$). Cela signifie que ces deux packages sont fortement liés l'un à l'autre. En ce qui concerne les dépendances internes et externes, on remarque que les packages *Gui*, *Models* et *Utils* sont plus orientés vers l'externe, ce qui confirme notre perception que ces dépendances concernent en particulier ces trois packages.

Pour les métriques *OutDep* et *IncDep,* le package *Gui* a 36 dépendances sortantes. Or, le package *Utils* a 34 dépendances entrantes. Par conséquent, il est claire que la majorité des dépendances partent du package *Gui* vers le package *Utils*. Ce qui explique aussi le nombre élevé de *Ce* et *Ca*, comparé au reste des packages.

Finalement, il est important de préciser que le langage PatOIS dans lequel toutes ces métriques sont implémentées, permet la gestion des ensembles d'objets composés. Ainsi, nous pouvons récupérer même l'ensemble des classes ou des packages qui sont impliqués dans une métrique quelconque. Par exemple, nous pouvons afficher l'ensemble des classes qui sont impliquées dans un couplage efférent ou afférent d'un package en particulier. Ceci est très avantageux par rapport à la majorité des outils de calcul de métriques, dans la mesure où cela donne une très bonne visibilité pour le mainteneur. Bien que ces métriques soient très significatives par rapport à l'architecture des packages, elles jouent aussi un rôle important dans la plate-forme pour l'implémentation d'autres métriques plus complexes, via le langage PatOIS. Par conséquent, nous allons utiliser ces métriques pour définir les quatre métriques principales des packages.

CPL								
Package	**N(E)**	**TotalDep**	**InternalDep**	**ExternalDep**	**OutDep**	**IncDep**	**Ce**	**Ca**
cfparse	1	0	0	0	0	0	0	0
attributes	2	6	0	6	0	6	0	1
util	2	0	0	0	0	0	0	0
awt	2	1	0	1	1	0	1	0
help	2	15	0	15	15	0	2	0
io	15	158	92	66	8	58	1	5
lang	3	30	6	24	24	0	4	0
multilingual	3	20	0	20	6	14	1	5
V14	10	2580	1618	962	962	0	91	0
V14.nodes	93	3354	1212	2142	364	1778	2	6
V14.visitors	4	1184	4	1180	816	364	92	92
V15	10	3000	1806	94	1194	0	113	0
V15.nodes	115	4264	1580	2684	452	2232	2	6
V15.visitors	4	1494	4	1490	1038	452	114	114
process	1	24	0	24	24	0	2	0
reference	3	0	0	0	0	0	0	0
XML	2	12	12	0	0	0	0	0
PADL								
analysis	2	4	2	2	2	0	1	0
event	8	62	32	30	26	4	7	2
kernel	60	532	418	114	57	57	5	8
exception	2	53	0	53	0	53	0	15
path	3	37	8	29	29	0	3	0
PapyrusForTest								
papyrus	2	2	2	0	0	0	1	0
dataBase	2	2	2	0	0	0	0	0
gui	7	48	12	36	36	0	4	1
models	3	2	0	2	0	2	0	1
utils	4	38	4	34	0	34	0	4

TABLE 5.4.1 – Résultats des métriques complexes

5.4.2 Métriques basées sur d'autres métriques : Les Métriques principales des packages

Dans la littérature, un consensus général est établi autour de deux métriques principales pour l'évaluation de la qualité des packages dans les systèmes OO, à savoir, la cohésion et le couplage. Cependant, R. Martin a mis la lumière sur deux autres métriques complémentaires, pour l'évaluation de la qualité des packages d'un point de vue conceptuel, la stabilité et l'abstraction.

Les résultats de ces métriques permettront de valider un modèle OO, en terme de modularisation, avec les principes de cohésion, de couplage, de stabilité et d'abstraction. À cet effet, nous avons implémenté ces quatre métriques au niveau du langage PatOIS, en se basant sur les métriques primitives et les métriques complexes présentées dans la section précédente. Les résultats de test de ces métriques sur les applications CPL, PADL et PapyrusForTest sont représentés dans la *table 5.4.2* ci-dessous.

CPL				
Package	**Cohésion**	**Couplage**	**Instabilité**	**Abstraction**
cfparse	0	0	0	0
attributes	0	1	0	0
util	0	0	0	0
awt	0	1	1	0
help	0	1	1	0.5
io	0.58	0.42	0.1	0
lang	0.2	0.8	1	0
multilingual	0	1	0.1	0
V14	0.63	0.37	1	0.1
V14.nodes	0.36	0.64	0.25	0.02
V14.visitors	0	0.99	0.5	0.5
V15	0.6	0.4	1	0.1
V15.nodes	0.37	0.63	0.25	0.01
V15.visitors	0	1	0.5	0.5
process	0	1	1	0
reference	0	0	0	0.6
XML	1	0	0	0.5
PADL				
analysis	0.5	0.75	1	0.5
event	0.51	0.85	0.78	0.25
kernel	0.78	0.97	0.38	0.98
exception	0	0.71	0	0
path	0.2	0.92	1	0.33
PapyrusForTest				
papyrus	1	1	1	0
dataBase	1	1	0	0
gui	0.25	0.91	0.8	0
models	0	0.5	0	0
utils	0.1	0.89	0	0

TABLE 5.4.2 – Résultats des métriques de packages

La cohésion

La métrique de cohésion permet d'évaluer la qualité d'un package par rapport à ses dépendances internes. D'après le principe CCP, un package doit contenir des classes qui sont en relation entre elles. Par conséquent, plus le taux de cohésion est élevé plus la qualité du package

est bonne.

D'après la *table 5.4.2*, le package XML a un taux de cohésion maximal, $(Cohesion(XML) = 1)$. Ceci est justifié par le fait que l'ensemble de dépendances de ses entités est en interne. Les packages qui ont le nombre de dépendances totale assez bas, par rapport aux packages principaux, ont le plus mauvais taux de cohésion, 0. Ceci peut s'expliquer par le fait que la majorité de ces packages n'ont pas de dépendances internes entre leurs entités. Cependant, deux packages en particulier attirent l'attention. Les packages *V14.visitors* et *V15.visitors* ont un taux de cohésion de 0. D'après l'analyse que nous avons effectuée sur les classes de ces deux packages, on note que ces classes sont des patrons visiteurs conçus pour parser des composantes différentes représentées sous la forme de plusieurs objets dans les packages *V14.Nodes* et *V15.Node*. Il est évident que les classes des packages *V14.visitor* et *V15.visitor* auront des relations de dépendances externes fortes avec les objets des packages *V14.Node* et *V15.Node*. Ainsi, nous pouvons conclure que le résultat de la cohésion pour ces deux packages est valide d'après leur conception dans le programme CPL. Par conséquent, ces deux packages sont mal structurés en terme de dépendances internes et non valides suivant le principe CCP.

Dans l'application PADL, le package kernel a le plus grand taux de cohésion, $(Cohesion(kernel) = 0.78)$. Ceci s'explique par le nombre de dépendances internes qui représente une grande partie des dépendances totales de ces package, $(InternalDep(kernel) = 418$ et $TotalDep(kernel) = 532)$. Ainsi, le niveau de cohésion de ce package est staisfaisant. Cependant, on note que le package exception a la plus faible cohésion, $(Cohesion(exception) = 0)$. Nous avons déjà prédit cela dans la section précédente, vu que nous avons remarqué que toutes les dépendances de ce package sont externes.

Concernant l'architecture de l'application PapyrusForTest, le package *models* a un taux de cohésion de 0. Ce package a deux dépendances externes entrantes vers ses entités sans avoir de dépendances internes. Par conséquent, il est tout a fait légitime qu'il ait un mauvais taux de cohésion. D'autre part, *Gui* et *Utils* ont un taux de cohésion assez bas à cause du nombre de dépendances externes qui représentent la majorité de leurs dépendances totales (voir *table 5.4.1*). Les packages *Papyrus* et *DataBase* ont un taux de cohésion maximal vu qu'ils n'ont pas de relations externes avec le reste des packages.

La constatation générale que l'on peut tirer à partir des résultats de la métrique de cohésion dans le *table 5.4.2*, est que l'architecture de plusieurs packages dans l'application CPL est mauvaise d'un point de vue cohésion, en particulier, les deux packages *V14.visitors* et *V15.visitors*. Ce-

pendant, dans le cas *PappyrusForTest*, la cohésion de l'ensemble des packages est satisfaisante, en générale, à l'exception des deux packages *Utils* et *Gui*, car ils ont un taux de cohésion en dessous de la moyenne, ($Cohesion(Utils) = 0.1$ et $Cohesion(Gui) = 0.25$). Les résultats du package *models* ne sont pas pris en considération vu qu'il n'a pas de dépendances internes. D'après les résultats, nous avons constaté que plus le programme est large en terme de classes et de fonctionnalités, plus le résultat a du sens par rapport à l'aspect que nous voulons mesurer par la métrique de cohésion.

Le couplage

La métrique de couplage permet d'évaluer un package par rapport à ses dépendances externes avec le reste des packages du système. D'après la *table 5.4.2*, la majorité des packages, pour le cas CPL, ont un taux de couplage au dessus de 0.5, en particulier les principaux packages de l'application. Les packages dont le taux de couplage est moyen, nécessitent des analyses supplémentaires par d'autres métriques, telles que la cohésion, la stabilité et l'abstraction.

Les packages *attributes*, *awt*, *help*, *multilingual* et *process* ont un taux de couplage qui avoisine le 100 %, malgré qu'ils n'ont pas un grand nombre d'entités. Ceci est certainement un défaut de conception au niveau du modèle OO. Une restructuration de ces packages est recommandée. En revanche, les développeurs peuvent toujours avoir un mot à dire, s'il y a des raisons sémantiques derrière leurs choix. Par conséquent, la décision de restructurer ces packages leur revient.

D'autre part, les packages *V14.visitors* et *V15.visitors* ont un taux de couplage de 0.99 et 1 respectivement. Ceci s'explique par le fait qu'ils ont un grand nombre de dépendances externes par rapport aux dépendances internes. De plus, et tel que nous l'avons expliqué dans la section précédente, la sémantique derrière leur structuration c'est d'isoler les différents patrons visiteurs dans ces packages, des composants visités, ce qu'a été confirmé par les résultats de la cohésion aussi. Ainsi, les résultats de couplage sont en accord avec les résultats de la cohésion ainsi que le modèle conceptuel réel de ces deux packages. Donc, on peut conclure d'après les résulats de couplage que ces packages sont de mauvaise qualité en terme de dépendance externe. On peut donc les considérer comme des packages cibles dans un processus de restructuration.

D'après les résultats de couplage dans la *table 5.4.2*, On note que l'ensemble des packages de l'application PADL, ont un mauvais taux de couplage au dessus de 0.71. Le package kernel, a le plus mauvais résultat au niveau de la métrique de couplage ($Couplage(kernel) = 0.97$).

Ce package est le plus grand parmi les packages de l'application, d'un point de vue taille ($N(E) = 60$). Ainsi, il est clair que ce package n'est pas bien structuré en terme de couplage.

Dans le cas de l'application *PapyrusForTest*, les packages *Papyrus* et *DataBase* ont un nivau de couplage nul. Ces deux packages ne dépendent pas des autres packages car ils ne contiennent que les configurations primaires de l'application, à savoir, le lancement de l'interface graphique et les connexions avec la base de données ainsi que la création des tables dans la base.

D'autre part, le package *Models* a un taux de couplage maximale ($Couplage(Models) = 1$). Cela signifie que toutes les dépendances de ce package sont externes. Les packages *Gui* et *Utils* ont un taux de couplage moyen de 0.75 et 0.89. Dans ce cas, la restructuration de ces deux packages n'est pas systématique car ils représentent respectivement les deux couches vue et contrôle du modèle MVC, adopté dans l'architecture de cette application. Par conséquent, une éventuelle restructuration risque de perturber la sémantique adoptée par le développeur pour l'architecture des packages dans l'application.

La métrique de couplage est très importante dans le processus d'évaluation de la qualité des packages. Elle permet d'évaluer un package en terme de ses dépendances avec les packages du système. Si le taux de cette métrique est très élevé, cela signifie que le package est très dépendant des autres packages, par conséquent, le package est très volatile face aux modifications. Pour mesurer le niveau de volatilité d'un package, on a introduit la métrique de stabilité pour estimer la fragilité des packages.

Instabilité

La métrique d'instabilité permet d'évaluer à quel point le modèle est valide suivant le principe SDP. D'après la *table 5.4.2*, on peut répertorier les packages des applications CPL, PADL et PapyrusForTest suivant deux catégories :

Les packages instables : ce sont les packages qui ont un taux d'instabilité maximale 1, à savoir, *awt*, *help*, *lang*, *V14*, *V15* et *Process* pour l'étude de cas CPL. Les résultats de la métrique d'instabilité deviennent ambigus lorsque le package a un nombre très faible de dépendances internes ou externes. D'après le modèle conceptuel de l'application CPL, nous avons remarqué qu'il y a une certaine hiérarchie entre les packages, *V14*, *V14.visitor* et *V14.node*. Et puisque on sait, d'après la définition de la métrique d'instabilité que nous avons présenté dans la section 4.2.3, que plus un package a des dépendances entrantes plus il est stable. D'après les résultats de

la table 5.4.2, ces packages ont un taux d'instabilité respective de 1, 0.5 et 0.25 ce qui confirme notre hypothèse qui consiste à dire que plus le package a un niveau plus bas dans un modèle OO, plus il est stable.

De même, on trouve les packages *analysis* et *path* dans l'application PADL ($I(analysis) = 1$ et $I(path) = 1$). Cela s'explique par le fait que l'ensemble des dépendances externes de ces deux packages sont des dépendances sortantes (voir *table 5.4.1*). Par conséquent, ces packages sont complètement instable. D'autre part, les packages *Papyrus* et *Gui* ont un taux d'instabilité élevé dans le cas *PapyrusForTest* ($I(Papyrus) = 1$ et $I(Gui) = 0.8$). En effet, la qualité de ces packages est dégradée dans la mesure où ils seront directement affectés dans le cas d'un éventuel processus de maintenance ou de modification dans le système analysé.

Les packages stables : ce sont les packages qui ont un taux d'instabilité quasiment nul tel que *cfParse*, *attributes*, *util*, *references* et *XML*, pour l'application CPL. D'autre part, le package exception est le seul package, complètement stable dans le cas PADL ($I(exception) = 0$). Il en est de même pour les packages *dataBase*, *models* et *utils* dans l'application *PapyrusForTest*. En effet, les résultats de la métrique d'instabilité pour ces packages est 0. Cela signifie que ces packages sont très responsables vu que les autres packages dépendent d'eux. En revanche, ces packages ne dépendent pas d'autres packages. Par conséquent, ils sont considérés stables, et n'importe quelle modification au niveau de ces packages peut entrainer des répercussions indirectes sur le reste des packages du système, qui ont des dépendances avec ces packages. Pour le reste des packages qui ont une valeur d'instabilité entre 0 et 1, plus le package est proche de la valeur 0, plus la qualité de ce package est bonne en terme d'instabilité, et inversement.

Pour conclure, la métrique d'instabilité est très importante pour l'évaluation de la qualité d'un package. En effet, elle permet d'identifier les packages qui sont très instables face aux changements. Le principe SDP vise à rendre le modèle le plus stable possible. Mais si la majorité des packages sont très stables, cela rend le modèle très figé, ce qui rend la tâche encore plus difficile aux mainteneurs, d'étendre le modèle.

Abstraction

La métrique d'abstraction permet de mesurer le niveau d'abstraction d'un package. Cette métrique peut être interprétée suivant deux angles. Premièrement, la métrique d'abstraction peut être interprétée individuellement pour évaluer le nombre d'entités abstraites par rapport aux

entités concrètes. Deuxièmement, cette métrique peut être interprétée proportionnellement à la métrique de stabilité.

D'après la *table 5.4.2*, les packages de l'étude de cas CPL, *help*, *V14.visitors*, *V15.visitors*, *references* et *XML* ont un niveau d'abstraction supérieur à 50%. Cela signifie que la moitié des classes de ces packages sont soit des classes abstraites, soit des interfaces, vu que le programme CPL est écrit en *Java*. De même, on trouve dans le cas PADL, le package *kernel* qui a le plus grand taux d'abstraction qui avoisine 100%, ($A(kernel) = 0.98$). Le reste des packages ont un taux d'abstraction moyen, sauf le package exception qui a un taux d'abstraction nul,($A(exception) = 0$). D'autre part, le reste des packages ont un niveau d'abstraction quasiment nul. De même, on note que tous les packages du cas *PapyrusForTest* ont aussi un niveau d'abstraction nul, ce qui signifie que ces packages n'ont aucune entité abstraite. Par conséquent, la structuration de ces packages n'est pas flexible et facile à modifier.

La métrique d'abstraction peut aussi être interprétée proportionnellement à la métrique d'instabilité. En effet, d'après le principe SAP, si le niveau d'abstraction augmente, le niveau d'instabilité doit baisser et inversement. Donc, si l'écart entre ces deux métriques est grand, le package est considéré comme non équilibré en terme de stabilité et abstraction. La *figure 5.4.1* décrit le graphe de l'écart entre les deux métriques A-I, et les points représentent l'ensemble des packages du cas CPL. D'après le graphe, on constate que les packages *V14.visitors*, *V15.visitors* et *Lang* sont les plus équilibrés. Mais par expérience, moins que la moitié des packages d'un projet arrive à satisfaire ces critères. Certains packages n'ont pas été pris en compte par cette métrique car ils ne sont pas connectés en externe.

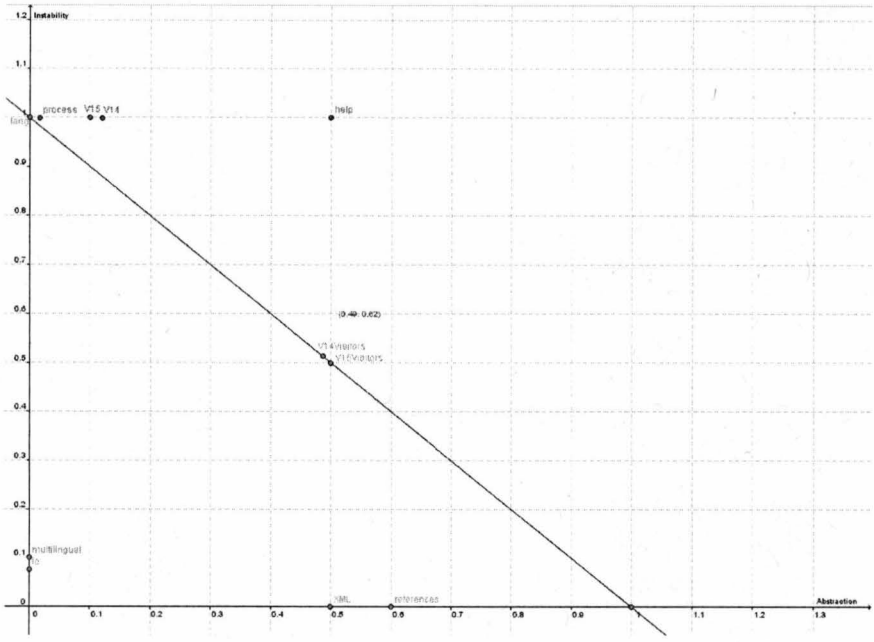


FIGURE 5.4.1 – Graphe de la métrique de distance A vs I

5.5 Graphe de dépendances

La majorité des outils de calcul des métriques, en particulier ceux qui traitent les dépendances inter-modulaires, contiennent des interfaces pour l'affichage des graphes de dépendances. En effet, la génération d'un graphe de dépendances est très pratique pour les développeurs. Il nous permet de visualiser la nature de dépendances d'une manière plus conviviale. À cet effet, nous avons intégré au sein du module de calcul des métriques, un mécanisme pour la génération du graphe de dépendances niveau packages et entités.

Pour la génération du graphe de dépendances, nous avons utilisé le langage DOT [2]. C'est un langage de description de graphes en format textuel. Après la génération du méta-modèle, un fichier DOT est créé automatiquement, suivant les dépendances et leurs directions. On présente dans la *figure 5.5.1*, une capture écran du graphe de dépendances, niveau packages et entités, pour le cas *PapyrusForTest*.

2. http ://fr.wikipedia.org/wiki/DOT_(langage)

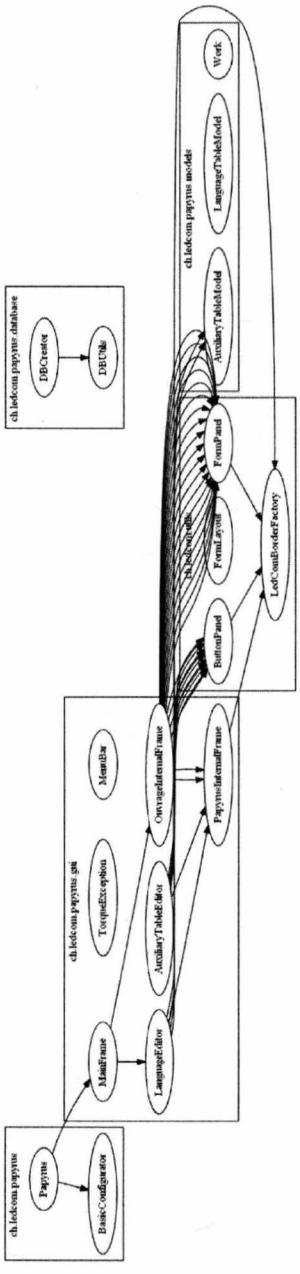


FIGURE 5.5.1 – Graphe de dépendances du cas PapyrusForTest

Le graphe de dépendances de la *figure 5.5.1* représente l'ensemble des packages du cas *PapyrusForTest* ainsi que les entités appartenant à ces packages. D'après la figure, on remarque que le nombre de dépendances ainsi que leurs natures, valide les résultats des métriques discutées

dans la section précédente. En effet, d'après le graphe, les packages *Gui* et *U*tils sont impliqués dans la majorité des relations de dépendances. Le package *Gui* dépends fortement du package *Ut*ils ; ceci est légitime dans le sens où le package *Utils* contient les fonctions de contrôle pour tous les composants graphiques disponibles dans le package *Gui.*

D'autre part, le package *dataBase* n'est pas relié au reste des packages du système. Cependant, le package *models* est impliqué dans deux relations de dépendances entrantes avec le package *Gui*, à travers l'entité *AuxiliaryTableModel.* L'avantage de cette représentation graphique est qu'elle permet de visualiser les dépendances internes et externes au sein même des packages. Cela facilite remarquablement la compréhension de la structuration des packages et la nature des dépendances qui les relient.

Conclusion

Bien que ces métriques nous donnent un aperçu assez clair par rapport à la qualité de la structuration des packages dans les systèmes OO, il demeure que les développeurs ont toujours un mot à dire par rapport à certain choix sémantiques. En revanche, les résultats des métriques que nous avons présentées dans ce chapitre, sont suffisants pour prendre une décision argumentée par rapport à la restructuration des packages.

Dans le but de tester l'efficacité des métriques implémentées, on a choisi d'étudier trois cas, *CPL, PADL* et *PapyrusForTest*, pour évaluer la structuration des packages en fonction de leur modèle OO. Le choix de ces trois applications n'était pas innocent. En effet, nous avons choisi trois applications qui ne sont pas trop complexes en terme du nombre de packages, car nous voulons vérifier la conformité de nos métriques avec le méta-modèle OO, établi durant la phase de *parsing*. Or, ceci nécessite une analyse manuelle en parallèle avec les résultats des métriques obtenus automatiquement.

En premier lieu, durant la phase de *parsing* et *mapping*, nous avons généré les instances méta-modèles des trois cas *CPL*, *PADL* et *PapyrusForTest*. La vérification des composantes dans chaque méta-modèle, se fait à partir des fichiers log tel que l'exemple dans la *figure 5.2.1* le montre. Par la suite, nous avons présenté les résultats obtenus des métriques implémentées au sein du module de calcul des métriques suivant les trois types de métriques, primitives, complexes et métriques basées sur d'autres métriques.

Les métriques primitives ont prouvé leur efficacité durant la phase des tests. En effet, elles ont joué le rôle d'une bibliothèque de fonctions pour l'implémentation d'autres métriques plus complexes, en langage PatOIS. Ceci était le principal rôle dédié à ces métriques. En se basant sur ces métriques, on a implémenté des métriques complexes. Les résultats de ces métriques étaient valides, efficaces et significatifs, dans la mesure où cela nous a permis de bien comprendre la structure interne et externe des packages avec des informations importantes sur la nature des relations entre ces derniers.

Les quatre métriques de packages, cohésion, couplage, instabilité et abstraction, représentent le troisième type de métriques gérées par le langage PatOIS. En effet, ces métriques se basent sur d'autres métriques décrites suivant le langage PatOIS. Les résultats de ces métriques ont confirmé ceux des métriques composés. Certains packages du cas CPL nécessitent une restructuration, vu que les résultats des ces métriques étaient mauvais, dans certains cas. Cependant, les résultats concernant le cas PayrusForTest sont satisfaisants, pour la majorité des métriques.

Finalement, nous avons présenté le graphe de dépendances niveau packages et entités, généré par la plate-forme après les phases de par*sing* et *mapping*. Le graphe est décrit suivant le langage DOT. D'après les tests effectués sur plusieurs cas et la comparaison avec quelques outils existants, nous avons constaté que les métriques sont suffisamment robustes pour le processus de restructuration.

Conclusion

La complexité des systèmes OO ne cesse d'augmenter, en particulier, durant la phase de maintenance, où on affirme que la plupart des modifications au niveau des fonctionnalités se produisent. Ceci est du aux changements impromptus des utilisateurs par rapport aux exigences. À cet effet, la modularisation dans le modèle conceptuel d'origine du système est perturbée. Suivant le modèle *ISO9125*, la modularisation est un attribut de qualité interne qui a une grande influence sur les caractéristiques externes de la qualité d'un logiciel[eAG01]. Par conséquent, les développeurs utilisent de plus en plus, des processus d'analyse et de mesure de la qualité des packages dans les systèmes logiciels, en particulier durant la phase de maintenance.

Alikacem et Sahraoui ont implémenté une plate-forme de description et d'extraction des métriques *"Boap FrameWork"*. Celle-ci est basée sur un langage de description pour le calcul des métriques, nommé PatOIS. En effet, elle permet aux utilisateurs d'adapter des métriques existantes suivant leurs besoins, ou d'étendre de nouvelles métriques d'une manière simple et flexible [AS09]. Le langage de description des métriques PatOIS, permet également la combinaison entre des métriques primitives (implémentées au sein du langage sous forme de bibliothèque des fonctions) et des éventuelles métriques prédéfinies avec le langage pour établir des nouvelles métriques non ambiguës. Des mécanismes tels que les itérateurs, les opérations et les ensembles sont disponibles, pour permettre la fusion de ces deux types de métriques. Cela dit, la plate-forme est limitée aux métriques niveau programmes, classes et méthodes.

Dans le cadre du travail décrit dans ce mémoire, nous avons adressé la problématique reliée à la compréhension et l'évaluation de l'architecture des packages dans les systèmes orientés objets. En effet, la nature abstraite des packages était la cause principale derrière le choix de l'approche, qui consiste à utiliser un méta-modèle OO pour la définition des métriques qui permettront de mesurer la qualité des packages suivant plusieurs critères. Durant notre étude de la littérature, nous avons remarqué que le manque de précision et l'ambiguïté des résultats obte-

nus à partir de ces métriques, étaient les principales lacunes souvent relevées par les chercheurs du domaine.

Pour répondre à ces lacunes, nous avons proposé dans ce document une extension de la plate-forme " *Boap FrameWork* " dans le but d'intégrer un module de calcul des métriques pour l'évaluation de la qualité des packages d'un programme OO. En effet, notre travail ne se résume pas qu'à l'aspect pratique, mais il inclut également un aspect théorique important. Ce dernier se manifeste par les analyses théoriques et les argumentations des choix adoptés pour la définition des métriques implémentées dans notre module, et l'extension de méta-modèle OO générique pour la gestion des informations relatives à la dépendance des packages.

Dans l'aspect théorique de notre travail, nous avons présenté une approche pour l'analyse de l'architecture des packages via un ensemble de métriques établies suivant les principes de conception proposés par R. Martin. Ces métriques sont utilisables pour caractériser un système suivant sa modularité. Elles vont des mesures simples aux métriques plus complexes. Parmi ces métriques, on trouve la cohésion, le couplage, l'abstraction et la stabilité. Celles-ci représentent les métriques principales des packages dans notre module. Tous d'abord, la cohésion est une métrique qui vise à évaluer la qualité des packages suivant ces dépendances internes. Or, le couplage permet de mesurer le niveau de dépendance d'un package avec le reste des packages dans le système. D'après les principes de cohésion et de couplage, un bon package est celui qui a un taux de cohésion élevé, contre un faible taux de couplage. D'autre part, la métrique de stabilité permet de prédire à quel point un package peut être impliqué face aux changements dans un éventuel processus de maintenance. Cependant, l'abstraction permet de mesurer à quel point un package peut être flexible face aux extensions. D'après les principes de stabilité et d'abstraction un certain équilibre doit être présent entre ces deux métriques pour conclure si le package est bon ou pas.

Dans l'aspect pratique de notre travail, nous avons décrit dans ce document les différentes étapes d'implémentation réalisées pour l'extension de la plate-forme. Tout d'abord, nous avons implémenté des nouvelles fonctionnalités au niveau du module de *parsing* et *mapping*, pour la représentation des informations sémantiques nécessaires pour le calcul des métriques des packages. D'une façon générale, nous croyons que chaque classe qui utilise un composant d'une autre classe, représente une relation de dépendance entre ces deux classes. Par conséquent, c'est une relation de dépendance entre leurs packages respectifs. Une relation de dépendance peut être externe dans le cas où les deux classes appartiennent à deux packages différents. En

revanche, une relation de dépendance est interne si les deux classes appartiennent au même package.

Au niveau du module évaluateur du langage PatOIS, nous avons implémenté plusieurs métriques primitives qui jouent le rôle d'une bibliothèque des fonctions pour la description d'autres métriques plus complexes, par exemple, la cohésion, le couplage, la stabilité et l'abstraction. En effet, elles permettent de mesurer les différents types de dépendances, niveau classe et package. Par la suite, nous avons implémenté les quatre métriques principales des packages avec le langage PatOIS en utilisant ces métriques.

Nous aurions pu implémenter les métriques plus complexes des packages de la même manière que les métriques primitives, mais cela aurait été un mauvais choix. En effet, la force du langage PatOIS c'est qu'il nous permet d'implémenter des métriques très complexes plus facilement que si on les implémente directement avec le langage du programme OO à analyser. De plus, il donne plus de flexibilité pour les utilisateurs de définir leurs métriques en adéquation avec leurs propres besoins. Par exemple, nous avons implémenté la métrique de cohésion avec le langage PatOIS suivant une formule bien précise. Or, un autre utilisateur peut définir la même métrique mais suivant une autre formule qu'il juge plus adéquate à ses besoins. Finalement, nous avons implémenté au sein de l'outil de la plate-forme les fonctions nécessaires pour la génération des graphes de dépendances, niveau classe et package. Cela donne une meilleure visibilité pour l'utilisateur, pour identifier les relations entre les packages d'un système OO.

Des études de cas ont été considérées pour vérifier l'efficacité de nos métriques, et cela nous a donné des résultats au niveau attendu. Par exemple, nous avons pu identifier les packages qui ne sont pas valides suivant les principes de conception CCP, CRP, SDP et SAP, dans les cas CPL, PADL et PapyrusForTest.

Actuellement, plusieurs pistes de travaux futurs peuvent être envisagées. On pense notamment à la possibilité d'étendre le module pour inclure des fonctionnalités de restructuration des packages. En effet, le langage PatOIS permet également la description d'un ensemble des règles qui se basent sur les résultats des métriques pour s'assurer de la conformité des packages par rapport à un certain patron de conception. Autrement dit, les principes de conception que nous avons discutés dans ce travail, peuvent être implémentés via ces règles, ce qui permettra d'automatisé la totalité du processus d'évaluation de la qualité des packages dans les systèmes OO. Une autre idée consisterait à enrichir le module par d'autres métriques primitives, dans le but de donner plus de flexibilité aux utilisateurs de la plate-forme, pour la description d'autres

métriques plus complexes concernant les packages.