الجمهورية الجزائرية الديمقراطية الشعبية

**REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE**

وزارة التعليم العالي و البحث العلمي

**Ministère de l'Enseignement Supérieur et de la Recherche Scientifique**

جامعة العربي بن مهيدي– أم البواقي–

Université L'arbi Ben M'hidi – Oum El Bouaghi –

Faculté des Sciences Exactes, des Sciences de la Nature et de la vie

Departement De Mathematiques et Informatique

# THESE

Présentée pour l'obtention du grade de **DOCTORAT 3ème Cycle**

**En** : Informatique

**Option** : Ingénierie des systèmes distribués

## Amélioration des modèles de variabilité dans les lignes de produits logiciels pour une meilleure adoption

**Présentée et soutenue par** : Araar Imad Eddine

**Thèse soutenue publiquement le 18 Juin 2018**
**devant le jury composé de :**

| | | |
|---|---|---|
| Mme. Seridi-Bouchelaghem Hassina | Professeur – Univ. Annaba | Directrice de thèse |
| Mr. Mokhati Farid | Professeur – Univ. Oum El Bouaghi | Président |
| Mr. Benmohamed Mohamed | Professeur – Univ. Constantine | Examinateur |
| Mr. Bourouis Abdelhabib | MCA - Univ. Oum El Bouaghi | Examinateur |

# ملخص

طيلة عقود كثيرة أطلقت العديد من المنظمات مبادرات إعادة استخدام البرمجيات لتحسين إنتاجيتها ."خطوط المنتجات البرمجية" (SPL)، المسماة أيضاً "بالعائلات البرمجية"، تناولت هذه المشكلة عن طريق تنظيم تطوير البرمجيات باستخدام مجموعة من الخواص التي تشترك فيها مجموعة من المنتجات. إن البرمجيات المطورة من قبل هاته المنظمات تعتبر نقطة انطلاق جيدة وقاعدة صلبة لترحيل تلكم المنظمات إلى حلول خطوط المنتجات البرمجية . لاستغلال منتجات البرمجيات القديمة في تطوير خطوط منتجات برمجية جديدة يجب تحديد الخواص المميزة لكل برمجية في المقام الأول. للقيام بذلك، يعتبر الرماز المصدري لهاته الأنظمة المصدر الأكثر موثوقية و الذي يوثق معارف الخبراء المشاركين في تطوير هذه البرمجيات. بيد أنه لا مفر من أن تكون هذه البرمجيات قد تم تطويرها من قبل عدة مهندسين، و ربما باستخدام لغات و تقنيات برمجة مختلفة أيضا. باختصار، نحن نواجه مسألة التباين في الرماز المصدري. في هذه الأطروحة، نتولى تقييم فعالية نهج جديد لإستخراج خطوط المنتجات البرمجية من الرماز المصدري للأنظمة الغرضية التوجه .مساهمتنا الأولى تتعلق باستخراج قائمة من الخواص المنفذة في البرمجيات المتاحة وذلك بإستخدام تقنيات التصرف التلقائي، مع الحد من فقدان المعلومات. مساهمتنا الثانية تتمثل في التعرف على المتغيرات و القواسم المشتركة في مجموعات الخواص المستخرجة، مع تناول مشكلة التباين، و ذلك من أجل إعادة بناء هذه المجموعات على شكل خطوط منتجات برمجية. وقد كشف تقييم النهج المقترح لدينا باستخدام اثنتين من البرمجيات المفتوحة المصدر "Java جافا" عن نتائج مشجعة.

**كلمات مرشدة**: خطوط المنتجات البرمجية، مخططات الخواص، تحليل البرامج، الهندسة العكسية، التصرف التلقائي، تنوع أعضاء العائلات، تشابه الدلالية، فهم البرامج، الانطولوجيا، وردنت، التصنيف.

# RESUME

Pendant de nombreuses décennies, plusieurs organisations ont lancé des initiatives de réutilisation des logiciels pour améliorer leur productivité. Les lignes de produits logiciels (LdP) ont abordé ce problème en organisant le développement de logiciels autour d'un ensemble de fonctionnalités qui sont partagées par un ensemble de produits. Pour accélérer la migration d'une organisation vers une solution LdP, les anciens systèmes déjà développés dans cette organisation constituent un appui fort. Afin d'exploiter les produits logiciels existants pour la construction d'une nouvelle LdP, les fonctionnalités qui composent chacun des produits utilisés doivent être spécifiées en premier lieu. Pour ce faire, le code source des systèmes analysés représente la source la plus fiable qui capitalise sur les connaissances des experts impliqués dans le développement de ces systèmes. Il est inévitable, cependant, que les systèmes développés dans une organisation soient créés par différent programmeurs, utilisant éventuellement différents langages et techniques de programmation. Bref, on se trouve confronté à un problème d'hétérogénéité. Dans cette thèse, nous évaluons l'efficacité d'une nouvelle approche d'extraction de LdP à partir du code source des systèmes orientés objet (OO). Notre première contribution concerne l'extraction de la liste de fonctionnalités implémentées dans un système existant utilisant des techniques d'apprentissage automatique, tout en minimisant la perte d'information. Notre deuxième contribution consiste à identifier les variabilités et les commonalités dans l'ensemble des fonctionnalités extraites, tout en adressant le problème d'hétérogénéité, afin de configurer ces fonctionnalités en une LdP. L'évaluation de notre approche proposée en utilisant différentes applications Java open-source a révélé des résultats encourageants.

**Mots clés :** Ligne de produits logiciels, Model de fonctionnalité, Analyse de programme, Rétro-ingénierie, Apprentissage automatique, Variabilité, Similarité sémantique, Compréhension du programme, Ontologie, WordNet, Partitionnement.

# ABSTRACT

For many decades, numerous organizations have launched software reuse initiatives to improve their productivity. Software product lines (SPL) addressed this problem by organizing software development around a set of features that are shared by a set of products. In order to accelerate organizations' migration to a SPL solution, old systems already developed in these organizations represent a strong base. To exploit existing software products for building a new SPL, features composing each of the used products must be specified in the first place. To do this, the source code of the analyzed systems represents the most reliable source that capitalizes on the knowledge of experts involved in systems development. It is unavoidable, however, that such systems are created by different programmers, possibly using different programming languages and techniques. In short, we are confronted with heterogeneity issues. In this thesis, we evaluate the effectiveness of a new approach to extract a SPL from the source code of the object-oriented (OO) systems. Our first contribution concerns the extraction of the list of features implemented in existing systems using machine-learning techniques, while reducing the information loss. Our second contribution is to identify the variabilities and commonalities in the extracted feature sets, while addressing the heterogeneity problem, in order to reconfigure such feature sets into a SPL. The evaluation of our proposed approach using various open-source Java applications has revealed encouraging results.

**Keywords:** Software product lines, Feature model, Program analysis, Reverse engineering, Machine learning, Variability, Semantic similarity, Software comprehension, Ontology, WordNet, Clustering.

# REMERCIEMENTS

*Je remercie vivement toutes les personnes qui m'ont aidé pendant l'élaboration de ma thèse et particulièrement ma directrice Madame le professeur Seridi Hassina, d'abord pour avoir accepté de superviser ce travail, et aussi pour sa disponibilité, son soutien constant et ses nombreux conseils tout au long du parcours.*

*Ce travail n'aurait pas été possible sans le soutien de l'Université Larbi Ben M'hidi d'oum El Bouaghi, qui m'ont fourni tous les moyens nécessaires pour élaborer ma thèse. Je remercie les membres responsables de la formation doctorale auquel je m'estime chanceux d'avoir suivi cette formation.*

*Mes sincères remerciements au Professeur Mokhati Farid, de l'université d'Oum El Bouaghi, qui a accepté de présider le jury, ainsi qu'aux autres membres de jury, le docteur Bourouis Abdelhabib, de l'université d'Oum el Bouaghi, et le professeur Benmohamed Mohamed de l'université de constantine.*

*Je remercie aussi celles et ceux qui me sont chers et que j'ai quelque peu abandonnés ces derniers mois pour achever cette thèse. Leurs encouragements m'ont accompagnée tout au long de ces années. Je remercie tout particulièrement mes amis Boutchicha Khaled et Mazouz Mihoub qui ont su être présents à tout instant.*

*Je suis sincèrement redevable à mon père, Araar Salah, pour son soutien moral et matériel et sa confiance dans mes choix. J'ai aussi une pensée toute particulière pour ma mère, qui a été touchée par la maladie d'Alzheimer juste au début de ma*

*formation doctorale. Je voudrais rendre hommage à son grand courage, à la mère qu'elle est, celle qui m'a donné la vie et qui a fait de moi ce que je suis aujourd'hui.*

*Je remercie aussi tous les autres membres de ma famille, et plus particulièrement mon frère, Araar Yassine, qui a été à mes côtés dans les moments les plus difficiles et quand j'avais des difficultés à croire en moi-même.*

*Je remercie aussi à ma femme, qui a été présente depuis le début de ce parcours et qui, sans elle dans ma vie, cette réussite n'aurait pas le même goût.*

*Enfin, et dans le désordre le plus total, merci à quelques amis qui ont compté ces dernières années : Toutou, Karim, Madjid, Youyou, Mimou, Hakou, Ali, Ramzi, Hocine …*

# DEDICACES

*A mon cher père*

*Et ma chère mère*

# TABLE DES MATIERES

# LISTE DES FIGURES

# LISTE DES TABLEAUX

# Chapitre 1: INTRODUCTION GENERALE

La productivité du logiciel n'a cessé d'augmenter depuis 1970, sans pour autant pouvoir combler l'écart entre les exigences imposées à l'industrie du logiciel et ce que l'état de la pratique peut offrir. De nos jours, pendant que les coûts des logiciels représentent encore une proportion croissante du coût des systèmes informatiques, et que les défauts des logiciels continuent d'être responsables de nombreux échecs coûteux, rien qu'une amélioration radicale de la qualité des logiciels et la productivité du développement ne sauvera l'industrie du logiciel de son état de crise perpétuelle. A ce problème, le développement des logiciels par réutilisation a été présenté, depuis les années 1960, comme la seule approche pratique et réaliste qui peut fournir de telles améliorations à court terme. Depuis les années 1970, Plusieurs organisations ont lancé des initiatives de réutilisation de logiciels. Cependant, les résultats varient largement, et dans les cas où la réutilisation a réussi, l'approche utilisée était parfois inimitable, inflexible, ou les deux, et ses avantages sont non mesurables.

Les lignes de produits logiciels (LdP), aussi appelées les familles de logiciels, représentent une approche très efficace et fructueuse pour planifier la réutilisation d'artéfacts logiciels dans une organisation. Le but de cette approche est de se focaliser sur la conception, le développement et la maintenance d'une gamme de produits au lieu de considérer un seul logiciel à la fois. Cette démarche permet donc de minimiser le cout et le temps de réalisation de produits de sorte que lorsqu'on on développe un artéfact on ne le fait qu'une fois pour toute une série de produits, sachant qu'un artéfact peut être n'importe quel élément de développement, allant des descriptions textuelles des besoins jusqu'aux données de test. Analogiquement, lorsqu'on effectue un changement sur un artéfact, ce changement sera donc appliqué à tous les produits qui disposent de cet artéfact. De plus, dans le contexte des LdP, les produits membres d'une famille possèdent des caractéristiques en commun tandis qu'ils varient dans d'autres. En d'autres termes, les produits possèdent des variabilités et des commonalités. La variabilité et sa gestion sont donc deux concepts clé qui différencient l'ingénierie des LdP de l'ingénierie des systèmes individuels. Les facteurs de variations peuvent être techniques (ex. la plateforme de déploiement), commerciaux (ex. plusieurs versions d'un produit), ou culturel (ex. produits appropriés à un payé cible).

La notion de LdP n'est pas totalement nouvelle car David Parnas [Parnas, D.L. 1976] a déjà étudié les familles de programmes dès 1976. Cependant, ce paradigme n'a émergé comme une approche à part du génie logiciel qu'à partir des années 1990. Motivé par les problématiques de ce nouveau paradigme, plusieurs chercheurs ont essayé de trouver des réponses satisfaisantes pour améliorer la qualité et la

productivité des LdP. Néanmoins, ces problématiques ne sont encore que partiellement maitrisées.

## 1.1 Motivations

Poussées par les exigences imposées à l'industrie du logiciel, plusieurs organisations ont choisi de se convertir aux solutions LdP. Bien qu'il existe plusieurs stratégies pour migrer une organisation vers une LdP [Krueger, C.W. 2002], la plupart des chercheurs sont aujourd'hui intéressés par les approches extractives pour accélérer et augmenter l'adoption des solutions LdP (plus de détails dans la section 2.1.6 ). A ce propos, les systèmes existants représentent des candidats potentiels pour être reconfigurés en LdP, utilisant des techniques de réingénierie.

## 1.2 Contributions

La thèse que nous défendons s'articule autour de deux contributions essentielles : (1) la génération d'un catalogue de fonctionnalités pour un système logiciel existant et (2) la configuration d'un ensemble de systèmes logiciels existants en une LdP.

La première contribution de cette thèse concerne la l'extraction de l'ensemble de fonctionnalités d'un système logiciel à partir de son code source. Comparée aux approches existantes pour la dérivation des fonctionnalités, la nouveauté de notre approche proposée est qu'elle fournit un catalogue générique et réutilisable de caractéristiques fonctionnelles pour une variante logicielle, au lieu de générer un seul modèle de fonctionnalités qui est spécifique à un (ou plusieurs) système analysé. Nous utilisons un algorithme de partitionnement avec chevauchement afin de minimiser la perte d'information. Dans notre l'approche proposée, les éléments de programmes Java constituent l'espace de recherche initial. En effectuant une analyse statique sur le système cible, nous définissons une mesure de similarité qui permet de passer par un processus de partitionnement. Le résultat est un ensemble de partitions d'éléments dont chacune représente une implémentation d'une caractéristique fonctionnelle. Le nombre de partitions est calculé automatiquement au cours du processus d'extraction ce qui diminue l'implication de l'expert.

La deuxième contribution concerne la configuration d'un ensemble de variantes logicielles en une LdP. La nouveauté de cette approche proposée et la possibilité d'extraire une ligne de produits logiciels à partir d'un ensemble de variantes qui sont hétérogènes au niveau du code source. Pour ce faire, nous dérivons d'abord les listes de fonctionnalités qui participent à la configuration de chaque variante. Par la suite, nous utilisons des techniques de traitements de langages naturels, une ontologie, et un nouvel algorithme proposé de partitionnement afin de remédier au problème d'hétérogénéité. Le résultat est une liste de caractéristiques fonctionnelles communes et variables des variantes analysées. Cette approche

nécessite une implication minime de l'expert au cours du processus d'extraction ce qui favorise son automatisation.

## 1.3 Plan du document

Ce manuscrit est organisé en quatre parties :

- La première partie présente l'état de l'art ; elle est divisée en deux chapitres : le chapitre 2 introduit le concept d'ingénierie des lignes de produits logiciels, ses objectifs et ses principes. Ce chapitre rappelle aussi les principes d'ontologies et de rétro-ingénierie et leurs concepts qui seront abordés dans le reste de ce document. Le chapitre 3 dresse un état de l'art des travaux existants sur l'extraction des lignes de produits logiciels.
- La deuxième partie présente en deux chapitres notre contribution à l'extraction des LdP. Le chapitre 4 détaille notre approche proposée pour la dérivation d'un catalogue de fonctionnalités à partir du code source d'un système logiciel. Le chapitre 5 explique, ensuite, une adaptation des ontologies au problème d'extraction des LdP.
- La troisième partie, représentée par le chapitre 6, fournit une évaluation empirique de l'outil réalisé.
- En fin, La quatrième partie conclut ce document par une description de quelques issues de recherche et travaux futures proposés.

# Chapitre 2: CONTEXTE GENERAL ET CONCEPTS CLES

Dans ce chapitre, nous présentons le contexte général et les prérequis nécessaires pour comprendre nos contributions détaillées dans les chapitres ultérieurs. La section 2.1 introduit le paradigme des lignes de produits logiciels. La section 2.2 explique la technique de classification issue de la fouille de données et les mesures nécessaires à leur validation. La section 2.3 présente les concepts de base des ontologies et de calculs de similarités sémantiques. La section 2.4 explique des concepts de base liés à notre travail. Enfin, la section 2.5 conclut ce chapitre.

## 2.1 Les lignes de produits logiciels

Les lignes de produits logiciels (LdP), aussi appelées les familles de logiciels, représentent une approche très efficace et fructueuse pour planifier la réutilisation d'artéfacts logiciels dans une organisation. Le but de cette approche n'est plus de développer un seul logiciel à la fois mais plutôt de concevoir et de développer une famille de produits logiciels, tout en considérant les caractéristiques variables dans ces produits, afin de réduire considérablement les coûts de développement en termes d'effort et de temps.

### 2.1.1 Définitions

On trouve dans la littérature plusieurs définitions permettant de qualifier les LdP. Nous citons, donc, quelques définitions représentatives:

- « *Une ligne de produits logiciels se réfère à des techniques d'ingénierie pour la création d'une collection de systèmes logiciels similaires à partir d'un ensemble partagé de ressources logicielles en utilisant un moyen commun de production* » [Krueger, C.W. 2011].
- « *Une ligne de produits logiciels est une famille de produits conçus pour tirer profit de leurs points communs et variabilités prévues* » [Weiss, D.M. and Lai, C.T.R. 1999].
- « *Une ligne de produits logiciels se compose d'une architecture de la gamme des produits et d'un ensemble de composants réutilisables qui sont conçus pour être incorporés dans l'architecture de la gamme de produits. En outre, la gamme de produits consiste à des produits logiciels qui sont développés en utilisant les ressources réutilisables mentionnés* » [Bosch, J. 2000].
- « *Une ligne de produits logiciels est un ensemble de systèmes à forte composante logicielle, qui partagent un ensemble commun et géré de fonctionnalités, satisfaisant les besoins spécifiques d'un segment ou d'une mission d'un marché particulier et qui sont développés à partir d'un ensemble commun de ressources de base d'une manière prescrite* » [Clements, P. and Northrop, L. 2001].

Bien que les définitions citées ci-dessus adressent le même phénomène, elles semblent comporter quelques différences légères. La définition donnée par Krueger [Krueger, C.W. 2011] introduit implicitement le concept de la portée d'une LdP. Weiss et al. [Weiss, D.M. and Lai, C.T.R. 1999] évoquent directement la notion de variabilité. Clements et Northrop [Clements, P. and Northrop, L. 2001] quant à eux mentionnent l'enjeu économique en expliquant que les systèmes issus d'une LdP doivent répondre aux besoins d'un marché existant. Jan Bosch [Bosch, J. 2000] évoque, par contre, l'aspect technologique en mettant l'accent sur la notion d'architecture logicielle. Finalement, ces différences entre les définitions citées ci-dessus ne signifient pas que les concepts définis étaient entièrement différents. En effet, chacune des définitions aborde le même phénomène d'un angle différent. En résumé, une ligne de produits logiciels peut être vue comme un ensemble de produits spécifiques qui possèdent beaucoup de fonctionnalités communes (commonalités) ainsi qu'un ensemble de points où on peut introduire des variations (variabilités). Ces produits sont conçus d'une manière prédéfinie en réutilisant des artéfacts qui sont partagés par les membres de la famille, afin de répondre aux besoins spécifiques d'un segment particulier du marché. Ces artefacts peuvent être tout type de ressources qui permettent de développer un logiciel (spécifications, modèle, code, données de test, … etc).

Il existe deux stratégies qui décrivent la façon dont les produits issus d'une LdP sont définis [van der Linden, F. et al. 2007] : (1) la stratégie *dirigée par le consommateur*, et (2) la stratégie *dirigée par le producteur*. Dans une stratégie dirigée par le consommateur, les produits spécifiques sont principalement définis par les exigences des clients actuels et futurs. Les produits finaux sont individualisés aux besoins spécifiques du client. Nous appelons cette situation la personnalisation de masse. Une personnalisation de masse est la production à grande échelle de produits adaptés aux besoins des clients individuels [Davis, S.M. 1987]. Dans cette situation, les exigences spécifiques de produits individuels sont très difficiles à identifier à l'avance. Il est très important que la plate-forme de la LdP fournit une base flexible pour le développement ultérieur des produits. Dans la stratégie inverse, i.e. dirigée par le producteur, il est surtout à l'organisme producteur de définir les produits. Cela est généralement le cas lorsque les produits sont développés pour des marchés de masse; où chaque produit issu de la LdP est vendu à un grand nombre de clients (souvent des centaines de milliers).

Les définitions présentées ci-dessus ont introduit plusieurs concepts qui sont à la base de l'ingénierie des lignes de produits logiciels, et qui sont essentiels pour comprendre le reste du travail présenté dans ce manuscrit. Il est donc instructif de définir chacun de ces termes :

### *La réutilisation logicielle :*

Est « *Le processus de la mise en œuvre de nouveaux systèmes logiciels en utilisant les informations sur un logiciel existant* » [Kang, K. et al. 1990].

### *La portée :*

La portée (anglicisme : portfolio), aussi appelé le domaine d'application ou la gamme de produits, est « *une collection d'applications (logicielles) actuelles et futures qui partagent un ensemble de caractéristiques communes* » [Berard, E.V. 1993]. En d'autre terme, la portée définie l'ensemble des catégories des produits (et non pas tous les produits individuels) qui sont pris en charge par l'organisation [Pohl, K. et al. 2005]. « *la portée d'une ligne de produits logiciels est déterminée par les limites des capacités fournies par la collection de produits dans la ligne de produits* » [Krueger, C.W. 2011]. Ainsi, l'analyse de la portée d'une ligne de produits détermine quels sont les produits qui feront partie de la ligne de produits et quel sont les produits qui ne seront pas considérés.

### *Une fonctionnalité :*

Une fonctionnalité ou une caractéristique logicielle (anglicisme : feature) « *représente un aspect important ou distinct qui est visible pour l'utilisateur, une qualité ou une caractéristique du système* » [Kang, K. et al. 1990].

### *Les commonalités :*

Weiss [Weiss, D. 1998] définit les commonalités (anglicisme : *commonality*) comme suit : « *une liste d'hypothèses qui sont vraies pour tous les membres de la famille* ». Ainsi, les commonalités représentent les fonctionnalités qui font partie de chaque produit, dans une ligne de produits, dans exactement la même forme.

### *La variabilité :*

La variabilité (anglicisme : *variability*) définit *« la façon dont les membres de la famille peuvent varier*» [Weiss, D. 1998]. Donc, contrairement aux commonalités, les variabilités sont une liste d'hypothèses qui ne sont pas vrais pour tous les membres d'une famille de produits logiciels.

### *Les points de variation :*

Les points de variation (anglicisme *variation point*) sont les emplacements où les variabilités peuvent apparaitre. Les points de variation sont considérés comme des décisions retardées de conception [Gurp, J.V. et al. 2001]. Avant sa conception, un point de variation est dit « *implicite* ». Donc, les point de variations ne deviennent « *explicites* » que lors (ou après) leurs conceptions.

### *Les variants :*

Les variants (anglicisme : *variants*) représentent la liste des choix possibles pour un point de variation. Un variant peut être constitué d'une ou plusieurs entités logicielles qui collaborent afin de résoudre une fonctionnalité requise. Ayant une idée sur le concept de variant, on peut distinguer deux autre type de points de

variation : (1) un point de variation dit « *ouvert* » dont on peut ajouter des nouveaux variants à sa liste des variants associée, et (2) un point « *fermé* » dont aucun nouveau variant ne peut être ajouté [Gurp, J.V. et al. 2001].

***Temps d'instanciation:***

Le moment d'instanciation ou de résolution de la variabilité (anglicisme : *binding time*) est le moment exact où les décisions retardées de conception sont prises. Le moment d'instanciation est variable et peut survenir tout au long du cycle de vie du développement d'une application spécifique. A ce moment-là, un produit partiellement ou totalement instancié est créé [Krueger, C.W. 2004].

***Les variantes logicielles :***

Les variantes logicielles (anglicisme : *software product line variants/products*) sont une collection de produits logiciels similaires qui partagent certains artefacts (i.e. code source, exigence, etc.) et diffèrent dans d'autres, pour répondre aux demandes spécifiques des clients dans un domaine particulier.

### 2.1.2 Motivations

Il existe plusieurs aspects qui sont à l'origine de l'ingénierie des LdP. Ces aspects varient allant de ceux qui sont beaucoup plus orientés processus tels que le coût et le temps, les qualités du produit comme la fiabilité, jusqu'aux aspects relatifs à l'utilisateur final telle que la cohérence de l'interface utilisateur. L'évolution vers l'ingénierie des LdP est généralement basée fortement sur des considérations économiques.

L'amélioration des coûts et des délais de commercialisation sont fortement corrélés dans l'ingénierie des LdP: l'approche prend en charge la réutilisation à grande échelle au cours du développement des logiciels. Contrairement aux approches classiques de réutilisation, ceci peut atteindre jusqu'à 90% du logiciel. La réutilisation est de loin plus rentable que le développement. Ainsi, les coûts de développement et le délai de la mise sur marché peuvent être amplement réduits par L'ingénierie des LdP.

Malheureusement, cette amélioration ne vient pas gratuitement, mais nécessite un certain investissement initial supplémentaire pour, par exemple, construire des artefacts réutilisables, transformer l'organisation, …etc. il existe plusieurs stratégies pour faire cet investissement (voir section 2.1.6), mais le besoin sous-jacent d'un investissement persiste toujours. Le bon côté, cependant, est qu'un seuil de rentabilité est généralement atteint après environ trois produits, et parfois plus tôt (voir *Fig 2.1*).

Généralement, la réduction des coûts de développement est atteinte aussi bien qu'une réduction des coûts de maintenance. Plusieurs aspects contribuent à cette

réduction; notamment le fait que le montant global du code et de la documentation qui doivent être maintenus est considérablement réduit.

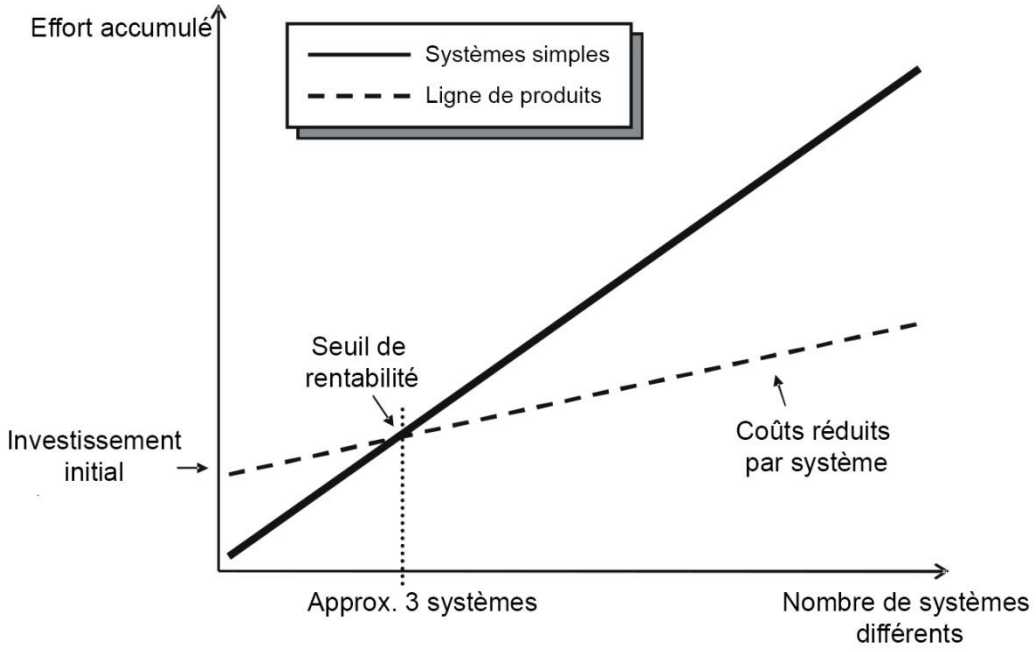



**Fig 2.1 : Les coûts de développement de n types de simples systèmes comparés à l'ingénierie des LdP [van der Linden, F. et al. 2007].**

L'ingénierie des LdP a également un fort impact sur la qualité du logiciel résultant. Un nouveau produit consiste, en grande partie, à des composants matures et éprouvés. On peut donc s'attendre à ce que la densité de défauts dans ces produits soit considérablement plus faible que celle des produits qui sont développés à partir de zéro. Cela conduit à des systèmes plus fiables et sécurisés. En conséquence, la sécurité est aussi impactée positivement. L'ingénierie des LdP peut également supporter l'assurance qualité, par exemple par le moyen de la simulation. Pour les systèmes embarqués, par exemple, la simulation permet de faire plusieurs tests approfondis, une chose qui serait impossible si on traite directement avec le produit final. Si le produit final et le produit sujet de simulation sont dérivés du même code, les simulations peuvent alors effectivement être utilisées comme base pour analyser la qualité du produit final. Bien que les arguments à propos du coût dominent généralement le débat d'ingénierie des LdP, la capacité de délivrer un produit d'une meilleure qualité est pour certaines organisations l'argument majeur, en particulier dans des domaines critiques où la sécurité joue un rôle important.

Au-delà des qualités du processus, l'ingénierie des LdP a un impact positif sur les aspects des produits tels que la facilité d'utilisation du produit final, par exemple en améliorant la cohérence de l'interface utilisateur. Ceci est réalisé en utilisant les mêmes artéfacts de base pour implémenter le même type d'interaction avec

l'utilisateur, par exemple en ayant un seul composant pour l'installation ou de l'enregistrement de l'utilisateur pour toute une gamme de produits au lieu d'avoir un composant spécifique pour chaque produit.

### 2.1.3 Historique

Le rêve de réutilisation massive des logiciels est aussi vieux que le génie logiciel lui-même. De nombreuses tentatives ou des initiatives de réutilisation de logiciels ont été faites, mais le plus souvent avec peu de succès. Ces initiatives de réutilisation étaient généralement fondées sur une approche de réutilisation à petite échelle ou bien ad-hoc (généralement au niveau du code; en outre le développement de nouveaux artéfacts était rarement basé sur une analyse systématique de la variabilité future).

Le concept de se concentrer sur un domaine spécifique comme base pour le développement d'artéfacts réutilisables n'a été introduit qu'un peu plus tard [Neighbors, J. 1986]. Cependant, dans la plupart du temps, les travaux menés dans ce contexte ont donné l'importance seulement au développement automatique de logiciels dans un domaine en utilisant des outils de génération. Cela a conduit au développement des langages spécifiques aux domaines (langages dédiés), mais, jusqu'à présent, ces travaux n'ont été adaptés pour le développement des systèmes à grande échelle.

Dans les années 1970, Parnas [Parnas, D.L. 1976] a déjà proposé le concept de familles de produits. Alors qu'il visait initialement la variabilité des caractéristiques non fonctionnelles, le concept de ligne de produits peut être retracé à ce travail.

Le concept de lignes de produits a été entièrement introduit dans les années 1990. L'une des premières contributions a été la description de la méthode FODA (pour *Feature-Oriented Domain Analysis*) [Kang, K. et al. 1990]. Vers la même époque plusieurs sociétés ont commencé à aborder la question de façon plus systématique. Par exemple, Philips a introduit la méthode de *Building-Block* dans le début des années 1990 [van der Linden, F. and Müller, J.K. 1995]. Ces premières approches ont été soutenues par des investissements massifs en Europe dans le domaine d'ingénierie des lignes de produits logiciels. A titre d'exemple, nous citons:

- Architectural Reasoning for Embedded Systems *ARES* (1995–1998);
- Product-line Realisation and Assessment in Industrial Settings *PRAISE* (1998–2000);
- Engineering Software Architectures, Processes and Platforms *ESAPS* (1999–2001) [van der Linden, F. 2002];
- From Concepts to Application in system-Family Engineering *CAFÉ* (2001–2003) [van der Linden, F. 2002];

- FAct-based Maturity through Institutionalization, Lessons-learned and Involved Exploration of System-family engineering *FAMILIES* (2003–2005) [FAMILIES project website 2005].

Ces projets ont soutenu la construction systématique d'une communauté de pratiques et de recherches en ingénierie des lignes de produits logiciels en Europe. Pendant la même période, le SEI (Software Engineering Institute) a soutenu le développement des lignes de produits logiciels aux Etats-Unis, et a publié plusieurs expériences industrielles prouvant sa réussite.

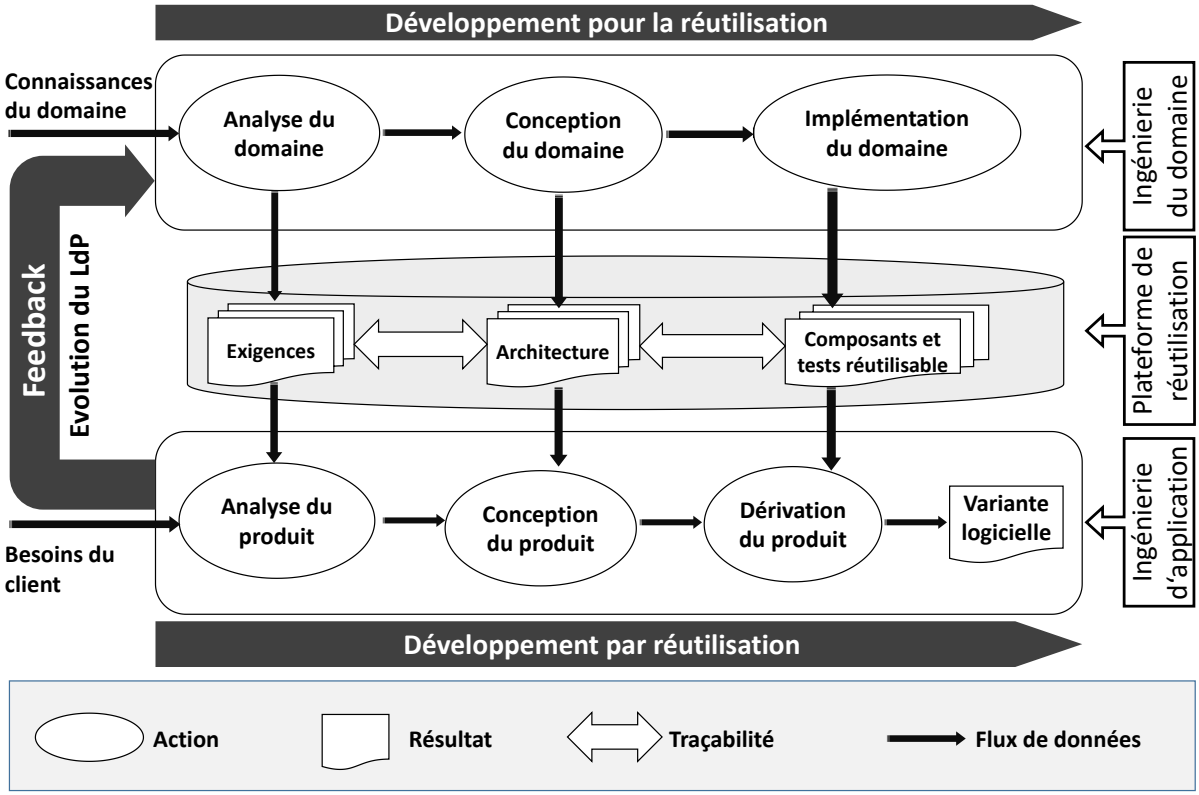



**Fig 2.2 : Les processus d'ingénierie du domaine et d'ingénierie d'application.**

## 2.1.4 L'ingénierie des lignes de produits logiciels

La différence majeure entre le développement conventionnel d'un seul système et l'ingénierie des LdP est le changement radical d'orientation: à partir d'un système individuel vers une LdP. Ce changement implique en particulier un changement de stratégie: migration d'une approche ad-hoc vers une vision stratégique d'un secteur d'activité. En effet, le but des lignes de produits est de favoriser une réutilisation logicielle effective et systématique. Pour ce faire, la mise en œuvre d'une ligne de produits est divisée en deux processus de développement parallèles [Weiss, D.M. and Lai, C.T.R. 1999, van der Linden, F. 2002] : un développement pour la réutilisation et un développement par réutilisation (voir *Fig 2.2*).

#### 2.1.4.1 L'ingénierie du domaine

Le but de ce processus de développement (développement pour la réutilisation) est de préparer une plateforme de réutilisation. Pour ce faire, nous distinguons trois activités : l'analyse, la conception et la réalisation du domaine. L'analyse du domaine a pour but d'étudier le domaine visé et de définir les commonalités et la variabilité de la ligne des produits. Les résultats de cette analyse vont être utilisés pour construire l'architecture de référence, i.e. une architecture du domaine, à partir de laquelle l'architecture de chaque produit sera dérivée. Une architecture de référence (architecture de LdP) est similaire à une architecture d'un système simple mais qui représente explicitement la variabilité identifiée durant l'analyse du domaine, utilisant des mécanismes de configuration, pour s'adapter aux besoins futures. Une architecture de référence fournit ainsi une structure commune de haut niveau pour tous les produits de la LdP. Enfin, la réalisation du domaine consiste à la conception détaillée et l'implantation d'artéfacts réutilisables des composants de l'architecture de référence. Les résultats de cette réalisation vont être réutilisés durant le processus suivant pour construire des produits spécifiques.

Ainsi, la plateforme se compose de tous les types d'artefacts logiciels (exigences, conception, code, tests, …) couvrant toutes les activités du développement des logiciels allant de l'analyse du besoin du domaine jusqu'à l'implémentation et les tests. Cependant, la qualité de ces artéfacts doit être assurée pour garantir la fiabilité des produits qui seront dérivés. En outre, des liens de traçabilité entre ces artefacts doivent être établis pour faciliter la réutilisation systématique et cohérente (par exemple, si on prend une exigence, on sera capable d'identifier toutes les implémentations et les cas de test relatifs à notre choix).

#### 2.1.4.2 L'ingénierie d'application

Ce processus consiste à dériver des produits spécifiques en se basant sur l'ensemble des besoins d'un client ou d'un segment de marché. L'architecture et les composants déjà construit dans l'étape précédente sont adaptés pour répondre aux exigences relatives aux produits individuels. Etant donné que ces artéfacts contiennent de la variabilité, on a donc besoin de décisions (variants) associés aux points de variation. Ainsi, la gestion des variabilités est une tâche cruciale qui doit être gérée et planifiée par des outils issus de l'ingénierie du domaine.

A ce stade du développement, plus de 90% du produit peut être obtenu par la réutilisation. Néanmoins, les parties de code spécifique doivent être développées pour implémenter les exigences spécifiques qui ne sont pas couvertes par les artéfacts du processus précédent. Ces nouveaux développements peuvent rejoindre la plateforme de réutilisation s'ils sont jugés assez génériques (*un feedback*), ce qui représente en quelque sorte une évolution de la ligne de produits.

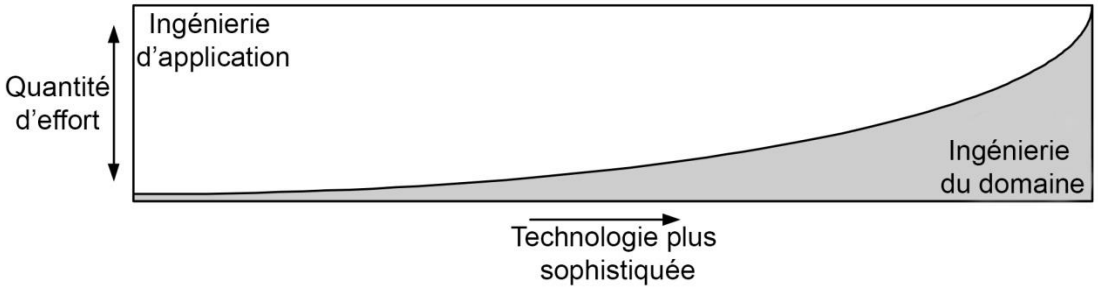



**Fig 2.3 : Rapport entre les efforts fournis dans l'ingénierie de domaine et l'ingénierie d'application [Deelstra, S. et al. 2005].**

L'essentiel des gains apportés par les LdP se retrouve alors durant la phase de l'ingénierie d'application. Deelstra *et al.* [Deelstra, S. et al. 2005] comparent les bénéfices apportés par l'ingénierie d'application par rapport à l'investissement requis pour créer une LdP : « L'idée derrière cette approche d'ingénierie de produits est que les investissements nécessaires pour développer les artéfacts réutilisables au cours de l'ingénierie de domaine sont compensés par les avantages pour calculer les produits individuels au cours de l'ingénierie d'application» [Deelstra, S. et al. 2005]. La figure *Fig 2.3* montre que plus l'investissement est élevé dans l'ingénierie du domaine, moins l'ingénierie de l'application demandera d'efforts. Cette figure montre aussi que le gain est proportionnel à l'investissement technologique engagé dans la LdP.

### 2.1.5 La variabilité

Lorsqu'on examine des produits, il devient évident que ces produits partagent certaines caractéristiques, bien qu'ils diffèrent dans d'autres. De même, dans le contexte des LdP [Weiss, D.M. and Lai, C.T.R. 1999], les produits possèdent des propriétés en commun, tandis qu'ils varient dans d'autres. Selon Weiss [Weiss, D. 1998], la variabilité définit la façon dont les membres d'une famille peuvent varier. Contrairement aux commonalités, les variabilités représentent donc une liste des hypothèses qui ne sont pas vrais pour tous les membres d'une famille de produits logiciels. La variabilité est un concept clé qui différencie l'ingénierie d'un système simple de d'ingénierie des LdP. Par conséquent, une bonne maitrise et gestion de la variabilité est susceptible d'augmenter la qualité et la productivité de la LdP. Cette

section aborde avec plus de détail le concept de variabilité, en expliquant ses dimensions, ses types, ainsi que les différentes tâches de sa gestion.

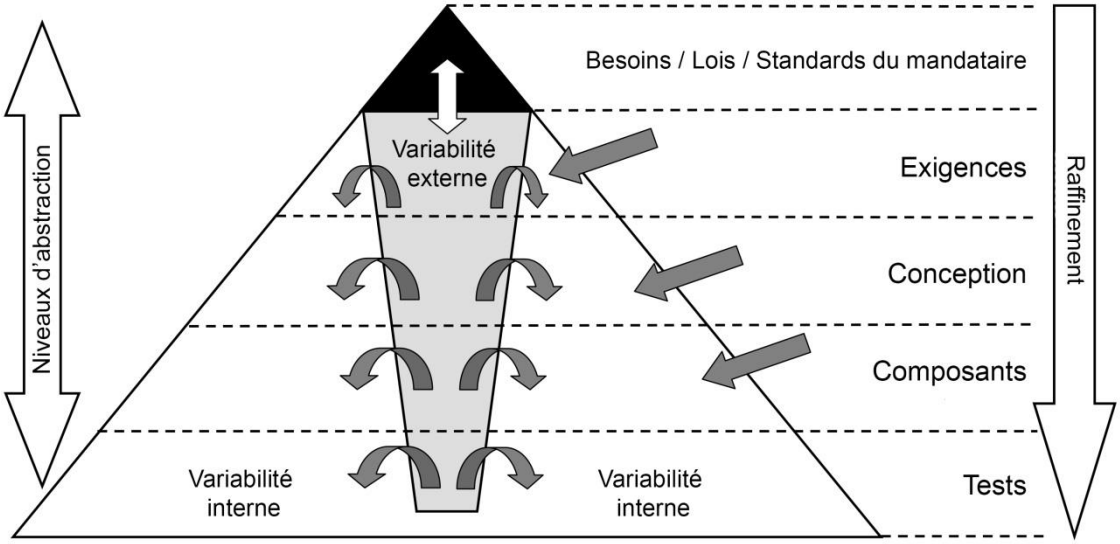



**Fig 2.4 : La quantité de variabilité dans chaque niveau d'abstraction [Pohl, K. et al. 2005].**

### 2.1.5.1 Dimensions et classification de la variabilité

Une première classification de la variabilité a été proposée, selon la dimension de la variabilité, et qui distingue ses deux types : (01) la variabilité dans le temps et (02) la variabilité dans l'espace [Bosch, J. et al. 2001, Pohl, K. et al. 2005]. En effet, en génie logiciel, l'évolution des artefacts avec le temps est une réalité incontournable, par exemple, quand ils doivent être adaptés en raison des progrès technologiques. Ce type de variabilité s'applique aussi bien à l'ingénierie d'un seul système qu'à l'ingénierie des lignes de produits logiciels. L'autre type de variabilité est spécifique aux lignes de produits logiciels. La dimension de l'espace concerne la variation entre plusieurs produits de la même famille. Autrement dit, c'est l'existence d'un même artéfact dans plusieurs produits avec des comportements différents.

D'autre études [Halmans, G. and Pohl, K. 2003, Pohl, K. et al. 2005] ont essayé de classifier la variabilité, vis-à-vis sa visibilité au client, en deux catégories: une variabilité *externe* (variabilité obligatoire) et une variabilité *interne* (variabilité technique). La variabilité externe représente la variabilité des artéfacts du domaine visible au client. En effet, les différentes exigences de client sont à l'origine de ce type de variabilité. Ainsi, chaque client doit pouvoir choisir parmi plusieurs options celles qui répondent le mieux à ses besoins. En ce qui concerne la variabilité interne, elle apparaît souvent lors du raffinement ou de la réalisation d'une variabilité externe (ou d'une autre variabilité interne. Voir *Fig 2.4*). La réalisation de chaque option offerte au client implique généralement plusieurs autres options plus fines aux niveaux inférieurs d'abstraction. Etant donné que le client est généralement intéressé par les décisions de haut niveau, les différentes possibilités

de réalisation ne doivent pas être communiquées avec lui. De plus, les problèmes techniques qui n'ont pas à être discutés avec le client peuvent aussi être à l'origine de la variabilité interne. Des exemples typiques de telles problèmes techniques sont l'implémentation, les problèmes de maintenance, la portabilité, …etc. Toutefois, la variabilité externe mérite une attention particulière car elle est visible pour les utilisateurs et concerne les exigences définies aux niveaux préliminaires de développement où les erreurs ou les imprécisions sont relativement peu coûteux et faciles à repérer et à corriger [Reinhartz-Berger, I. et al. 2011].

#### 2.1.5.2 La gestion de variabilité

L'ingénierie des lignes de produits logiciels vise à soutenir une gamme de produits. Ces produits peuvent satisfaire les demandes de différents clients individuels ou peuvent répondre entièrement aux besoins de différents segments du marché. De ce fait, la variabilité est un concept clé dans une telle approche. Au lieu de comprendre chaque système individuel tout seul, l'ingénierie des lignes de produits examine la gamme de produits dans son ensemble ainsi que la variation entre les différents systèmes. Cette variabilité doit être définie, représentée, implémentée, évoluée, …etc. En un mot, la variabilité doit être *gérée*, tout au long de l'ingénierie de la ligne de produits logiciels. Ainsi, la variabilité et sa gestion sont les principales caractéristiques qui distinguent l'ingénierie des lignes de produits logiciels d'autres approches de développement [Bosch, J. et al. 2001].

##### 2.1.5.2.1 La modélisation de variabilité

La modélisation de variabilité est la tâche la plus importante dans la gestion de variabilité et qui a été adressée par la plupart des recherches sur la gestion de variabilité dans les LdP [Chen, L. et al. 2009, Alves, V. et al. 2010]. La modélisation peut être utilisée comme un mécanisme pour définir et représenter la variabilité, ainsi que les interrelations entre les éléments qui composent une LdP, d'une manière contrôlée et traçable. Les techniques de modélisation de la variabilité ont été développées pour aider les ingénieurs à défier les complications de la gestion de variabilité. L'idée derrière ces techniques est de représenter la variabilité d'une manière assez explicite pour augmenter sa compréhension, afin d'automatiser sa gestion.

Une technique de modélisation doit avoir un langage bien défini pour représenter la variabilité. A ce propos, les modèles de fonctionnalités (MF) fournissent une description détaillée des commonalités et des variabilités, en précisant toutes les configurations valides de fonctionnalités. Les deux termes « modèle de fonctionnalités » et « diagramme de fonctionnalités » ont été utilisés dans la littérature (dans le contexte des LdP) de manière interchangeable, faisant référence à un modèle qui décrit la variabilité dans une LdP en termes de fonctionnalités. La modélisation de fonctionnalités est une approche relativement simple pour

modéliser les capacités d'un système logiciel, et qui a été introduite pour la première fois par Kang et al. [Kang, K. et al. 1990] dans leur méthodologie FODA (pour *Feature Oriented Domain Analysis*). Les MF ont été proposés donc comme étant une partie de cette méthodologie, et sont maintenant un moyen très répandu pour l'expression de la structure d'une LdP. Ils sont aussi utilisés comme une entrée de premier ordre lors de la dérivation des produits. Depuis la proposition de FODA, plusieurs extensions de la notation de FODA ont été introduites pour représenter la variabilité sans pour autant arriver à une notation unifiée. Les différents langages de modélisation de fonctionnalités peuvent être classés, selon le type de notation, en deux catégories : des langages graphiques et des langages textuels.

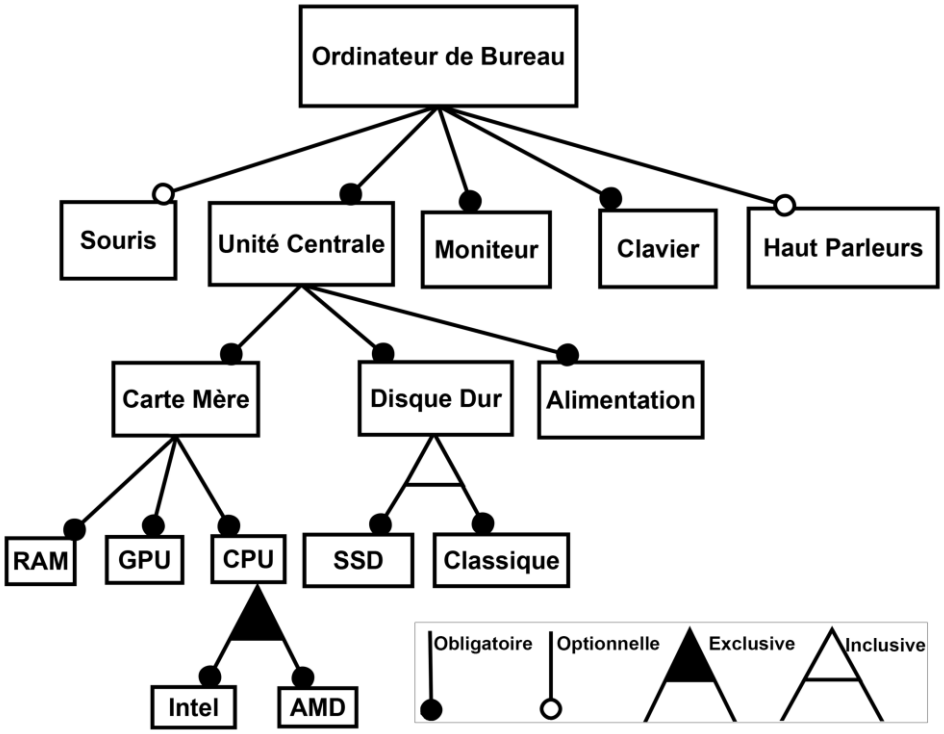



**Fig 2.5 : Exemple simplifié de MF pour une LdP d'ordinateurs de bureau.**

La figure *Fig 2.5* illustre l'exemple d'un simple MF dans lequel les fonctionnalités sont représentées sous forme de boîtes étiquetées et sont reliées par des lignes à d'autres fonctionnalités avec lesquelles elles se rapportent, formant collectivement une structure arborescente. Une fonctionnalité peut être classée comme: *obligatoire* si elle fait partie d'un produit à chaque fois que sa fonctionnalité ascendante est également sélectionnée, et *optionnelle* si elle peut ou non faire partie d'un produit à chaque fois que sa fonctionnalité ascendante fait partie. Les fonctionnalités peuvent être regroupées en: (01) *inclusives* pour désigner une relation dans laquelle une ou plusieurs fonctionnalités du groupe peuvent être sélectionnées (i.e. OR), et (02) *exclusives* indiquant que seulement une seule

fonctionnalité peut être sélectionnée (i.e. XOR). La fonctionnalité racine d'une LdP est toujours incluse dans n'importe quelle combinaison (i.e. configuration).

Une fois la variabilité identifiée, elle doit être limitée. En effet, le but n'est pas de fournir une flexibilité sans limites, mais de fournir juste assez de flexibilité pour répondre aux besoins actuels et futurs du système d'une manière efficace. Etant donné un intervalle des choix possibles dans les MF, seulement certaines combinaisons de décisions pour ces choix peuvent conduire à un produit utile et cohérent. Les restrictions sur les décisions relatives aux choix sont dénommées les contraintes. Un exemple d'une contrainte est que lorsqu'une fonctionnalité spécifique $f_1$ est incluse (resp. exclue) alors une autre fonctionnalité $f_2$ doit être aussi sélectionnée (resp. exclue) pour assurer le bon fonctionnement du produit. Lorsque ces contraintes sont formalisées explicitement dans un MF, la cohérence de la configuration peut être vérifiée (automatiquement) sans avoir testé le produit logiciel qui en résulte. La façon avec laquelle les techniques de modélisation de la variabilité définissent formellement les contraintes diffère considérablement d'une technique à l'autre allant d'une simple règle de type « inclut » et « exclut » jusqu'aux expressions algébriques avancées.

#### 2.1.5.2.2 Implémentation de la variabilité

Alors que la modélisation est une technique pour représenter la variabilité, les mécanismes de variabilité sont généralement considérés comme des moyens pour implémenter cette variabilité. Le choix du mécanisme d'implémentation de la variabilité est indépendant de la sa modélisation. Il s'agit d'une conséquence des décisions prises durant l'ingénierie du domaine. Le choix du mécanisme d'implémentation est dirigé par deux facteurs essentiel [Bosch, J. and Capilla, R. 2013]: (1) le niveau d'abstraction dans lequel le point de variation est exploré, allant de l'architecture jusqu'au code, et (2) l'étape dans le cycle de vie durant laquelle le point de variation est instancié (i.e. le temps d'instanciation).

Dans ce qui suit nous donnons quelques exemples pratiques et illustratifs des mécanismes de variabilités. D'autres mécanismes plus sophistiqués ont été cités dans [Svahnberg, M. and Bosch, J. 2000, Pohl, K. et al. 2005, Bosch, J. and Capilla, R. 2013].

***Les macros :***

Cette technique permet l'adaptation d'un composant lors de sa compilation. Avant la compilation, le compilateur lit un ou plusieurs fichiers contenant des définitions de macros. Une macro peut être n'importe quel fragment de code de programme et peut donc être utilisée pour implémenter un point de variation. Des fragments de code définis représentent des variants pour ce point de variation. Les macros sont remplacées par leurs définitions à chaque occurrence dans le code.

***La programmation orientée objet :***

Les langages de programmation offrent une multitude d'options qui peuvent être utilisées pour implémenter la variabilité. L'héritage dans les langages orientés objet est une technique standard pour étendre le comportement d'une classe mère en ajoutant un comportement supplémentaire aux classes descendantes. Cette technique permet de prendre en compte la variabilité en réutilisant les fonctionnalités en commun avec la nouvelle variante logicielles (ex. le comportement de la classe mère) et seulement l'étendre pour réaliser la variabilité. Un autre exemple se présente lorsque le point de variation est une méthode qui doit avoir une implémentation différente pour chaque variante logicielle.

***La programmation orientée aspect :***

la programmation orientée aspect [Kiczales, G. et al. 1997] est un paradigme pour la gestion de la séparation des préoccupations transversales et la décomposition d'un système en utilisant plus d'un critère. Plusieurs chercheurs ont essayé d'adopter les principes de ce paradigme pour implémenter la variabilité. En effet, l'utilisation du code basé sur un langage orienté aspect est susceptible de réduire l'effort fournit durant la programmation pour spécifier des temps d'instanciations des variabilités, afin de prendre en charge les changements dynamiques (i.e. les fonctionnalités du système qui peuvent être activées dynamiquement). De plus, une certaine pratique permet d'améliorer la maintenabilité des LdP.

***L'extension:***

Ce mécanisme permet à une application d'adapter dynamiquement son comportement en ajoutant de nouveaux modules de code durant son exécution en utilisant, par exemple, des plugins (ici le plugin représente un variant) afin de s'adapter à des besoins spécifiques.

#### 2.1.5.2.3 Traçabilité

Une fois une LdP initiale est développée, elle est susceptible d'être sujette à une évolution dans le temps. Cela nécessite non seulement d'ajouter de nouvelles variations, mais aussi d'étendre les fonctionnalités communes. Le principal défi ici est de savoir comment réaliser une telle évolution d'une manière efficace et avec le moins d'impact sur les produits existants et sur l'architecture de référence. A cette fin, il est nécessaire de garder une trace du flux d'informations depuis les expressions textuelles des besoins jusqu'aux produits individuels, et d'enrichir ce flux d'information avec les justifications des choix conceptuels spécifiques. Donc, en plus de modéliser les points de variation et de variants ainsi que leurs relations, les développeurs doivent aussi relier la variabilité définie dans le modèle de variabilité aux artéfacts logiciels spécifiés dans les documents textuels, dans d'autres modèles (ex. architecture), et dans le code (Voir *Fig 2.2*).

En effet, la gestion de variabilité diffère radicalement d'un niveau d'abstraction à un autre. Par exemple, la variabilité définie dans l'architecture ne définit pas la manière avec laquelle cette variabilité sera implémentée dans le code, vue qu'il existe plusieurs mécanismes de programmations qui peuvent être utilisés. De plus, la modification de la variabilité structurelle (celle qui est définie dans un MF) influe grandement les niveaux inférieurs.

La façon de documenter ces relations est donc de définir des liens de traçabilité entre le modèle de la variabilité et les artefacts de développement.

### 2.1.6 Les stratégies d'adoption

La migration d'une organisation vers une ligne de produits logiciels peut fonctionner en utilisant l'une des trois grandes stratégies d'adoption [Krueger, C.W. 2002]: *proactive*, *extractive* et *réactive*. Les sections suivantes fournissent plus de détails sur ces trois modèles.

#### 2.1.6.1 L'approche proactive

Avec l'approche proactive, l'organisation analyse, conçoit et met en œuvre une LdP pour soutenir tous les produits prévus (qui sont dans la portée du LdP), allant d'une analyse et une conception, jusqu'à l'implémentation du code source (commun ou spécifique).

L'approche proactive est similaire à l'approche en cascade pour les systèmes simples. Elle est convenable lorsque les exigences requises pour l'ensemble de produits prévus sont bien définies et stables. L'approche proactive exige beaucoup plus d'effort au début, mais cela diminue brusquement une fois que la LdP est achevée. Si le coût, le temps et l'effort sont prohibitifs au début, ou si le risque d'une mauvaise prédiction est élevé, l'utilisation d'une approche réactive est alors plus appropriée.

Pour employer une approche proactive on doit suivre les étapes ci-dessous :

1. Analyser le domaine et la portée pour déterminer la variation supportée par la LdP ;
2. Modéliser l'architecture de la LdP pour soutenir la dérivation de tous les produits prévus;
3. Concevoir les parties communes et les parties variantes du système ;
4. Enfin, mettre en œuvre les parties communes et les parties variantes du système.

Une fois la LdP a été mise en œuvre, tout ce qui reste est de créer des instances de produits en fonction des besoins. Avec l'approche proactive, si de nouveaux

produits sont requis, ils sont susceptibles d'être dans la portée existante de la LdP et peuvent être dérivés simplement.

#### 2.1.6.2 L'approche réactive

Avec l'approche réactive, les organisations développent leurs LdP d'une façon incrémentale. Cette approche est appropriée lorsque les exigences des nouveaux produits dans la LdP sont en quelque sorte imprévisibles. Les modèles de variabilités sont progressivement étendus en réaction aux nouvelles exigences. Cette approche incrémentale offre une transition plus rapide et moins coûteuse vers les LdP, étant donné qu'un nombre minimum de produits doit être incorporé à l'avance.

Pour employer une approche incrémentale on doit suivre les étapes ci-dessous :

1. Vérifier que les exigences d'un nouveau produit sont actuellement prises en charge par la LdP ;
2. Si le nouveau produit est dans la portée actuelle de la LdP, passer à l'étape 4 ;
3. Si le nouveau produit n'est pas dans la portée, étendre la portée pour inclure les nouvelles exigences;
4. Créer le produit.

#### 2.1.6.3 L'approche extractive

Avec l'approche extractive, l'organisation capitalise sur les systèmes logiciels existants par l'extraction de commonalités et de variabilités. Cette approche est plus appropriée lorsque l'ensemble de systèmes possède un nombre important de commonalités ainsi que quelques différences constantes.

En outre, il n'est pas nécessaire de procéder à l'extraction en utilisant tous les systèmes préexistants à la fois. Par exemple, on peut effectuer l'extraction, dans un premier temps, à partir d'un sous-ensemble de systèmes possédant un gain important, et utiliser le reste de systèmes progressivement en fonction de besoins. Par conséquent, une telle réutilisation de haut niveau des logiciels permet à une organisation d'adopter très rapidement l'approche LdP.

## 2.2 L'apprentissage automatique

Pour résoudre un problème sur un ordinateur, nous avons besoin d'un algorithme. Un algorithme est une séquence d'instructions qui devraient être exécutées pour transformer l'entrée en sortie. Cependant, il y a beaucoup d'applications pour lesquelles on n'a pas un algorithme mais seulement des données d'exemple (des observations) stockées dans des bases de données, et on veut que l'ordinateur (machine) extraie automatiquement des connaissances pour cette application. Par

exemple, on n'a pas un algorithme pour dire si un email est un spam ou non. Ce qui peut être considéré comme Spam change avec le temps et d'un individu à un autre. Néanmoins, on croit qu'il y a un processus qui explique les données qu'on observe, et que ces derniers ne sont pas complètement aléatoires. Il y a, Donc, certains « *patrons »* dans les données. Par conséquent, même si l'identification du processus complet peut ne pas être possible, on peut encore détecter certains *patrons* ou *régularités*, et construire une bonne et utile approximation. C'est là qu'intervient l'apprentissage automatique. Mais l'apprentissage automatique n'est pas simplement un problème de base de données ; c'est également un champ d'étude de l'intelligence artificielle. Alors que l'intelligence artificielle s'intéresse en général à reproduire la capacité cognitive de l'être humain par une machine, l'apprentissage automatique vise plus particulièrement la capacité d'apprentissage. Pour être intelligent, un système, qui est dans un environnement en cours d'évolution, devrait avoir la capacité d'apprendre. Si le système peut apprendre et s'adapter à de tels changements, le concepteur de système n'a pas besoin de prévoir et fournir des solutions (à la main) pour toutes les situations possibles [Alpaydin, E. 2010].

La Fouille de données, la science et la technologie de l'exploration des données, est une partie de l'ensemble du processus de découverte de connaissances dans les bases de données (KDD[1]). Elle est essentiellement utilisée pour dénicher des tendances ou des corrélations cachées parmi des masses de données, ou encore pour détecter des informations stratégiques ou découvrir de nouvelles connaissances, en s'appuyant sur les techniques d'apprentissage automatique et les méthodes statistiques [Rokach, L. and Maimon, O. 2008].

### 2.2.1 La segmentation de données

La segmentation de données, aussi appelée partitionnement, catégorisation, classification non supervisée (anglicisme : *clustering*), est une technique d'apprentissage automatique qui consiste à former des groupes (i.e. clusters) homogènes à l'intérieur d'un ensemble de données (i.e. une population). Pour cette tâche, il n'y a pas de classe à expliquer ou de valeur à prédire qui est définie a priori, il s'agit de créer des groupes homogènes dans la population. Il s'agit alors de trouver des régularités dans les entrées ; il existe une structure dans les données d'entrées telles que certains patrons se produisent plus souvent que les autres, et on veut voir ce qui se passe en général et ce qui ne le fait pas [Alpaydin, E. 2010]. Donc, Aucun expert n'est requis et l'algorithme de segmentation doit découvrir par lui-même la structure plus ou moins cachée des données. C'est seulement à la fin de l'apprentissage qu'un expert du domaine intervient, pour déterminer l'intérêt et la signification des groupes ainsi constitués.

[1] Knowledge Discovery in Databases.

Il y a plusieurs domaines de recherche où la segmentation a été appliquée avec succès, par exemple: la recherche d'information, la segmentation d'images, la classification des empreintes digitales, l'analyse des réseaux sociaux, … etc. Bien que plusieurs algorithmes de segmentation ont été proposés au cours des dernières années, la plupart d'entre eux construisent des clusters disjoints, i.e. ils ne permettent pas aux objets d'appartenir à plus d'un cluster. Cependant, il existe certaines applications où il est souvent le cas que des objets appartiennent à plus d'un cluster. Pour ces types d'applications, une segmentation avec chevauchent est utile et indispensable. Il existe certains algorithmes de regroupement qui ont été rapportés dans la littérature pour résoudre le problème de chevauchement de clusters, mais ils ont quand même des limitations qui pourraient réduire leur champ d'application ou leur utilité dans des problèmes pratiques. La section suivante introduit l'algorithme OClustR [Pérez-Suárez, A. et al. 2013], un algorithme de segmentation avec chevauchement qui s'est montré, selon les expérimentations menées par ses auteurs, plus performant que d'autres algorithmes qui adressent le même problème.

### 2.2.2 L'algorithme OClustR

L'algorithme OclustR [Pérez-Suárez, A. et al. 2013] est un algorithme de segmentation avec chevauchement qui est basé sur la théorie des graphes et qui s'exécute sur un graphe non orienté. Il passe par deux étapes principales: (1) l'étape d'initialisation, et (2) l'étape d'amélioration.

L'idée principale de la phase d'initialisation consiste à produire un premier ensemble $X$ de sous-graphes, i.e. ws-graphes, qui couvre le graphe ; chaque ws-graphe consiste en un cluster candidat. Par la suite, pendant la phase d'amélioration, un post-traitement est effectué sur les sous-graphes initiaux afin de réduire leur nombre et leur chevauchement. Pour ce faire, l'ensemble $X$ est analysé pour supprimer les ws-graphes qui sont considérés comme moins utiles. Ces derniers sont supprimés, en fusionnant les groupes de leurs sommets avec ceux d'un ws-graphe choisis.

Formellement, soit $O = \{o_1, o_2, \ldots, o_n\}$ un ensemble d'objets du graphe qui représentent des entités dans un domaine d'application. L'algorithme OClustR utilise comme entrée un graphe pondéré non orienté $\tilde{G}_\beta = (V, \tilde{E}_\beta, S)$ tel que $V = O$, et il existe une arête $(v, u) \in \tilde{E}_\beta$ ssi $v \neq u$ et $S(v, u) \geq \beta$, avec $S(o_1, o_2)$ est une fonction symétrique de similarité et $\beta \in [0,1]$ est un seuil défini par l'utilisateur ; chaque arête $(v, u) \in \tilde{E}_\beta$ est étiquetée par la valeur de $S(v, u)$.

### 2.2.3 Mesure d'évaluation

Après avoir obtenu un partitionnement d'éléments de données en clusters, on doit s'assurer que cette solution est valide, ce qui signifie qu'elle regroupe vraiment les

objets similaires et sépare les objets dissemblables. En particulier, la solution est valide lorsque les clusters contiennent des objets fortement corrélés. Cette étape de validation est fondamentale pour obtenir des résultats fiables de clustering. Ci-dessous nous introduisons les concepts liés aux mesures utilisées dans notre travail.

#### 2.2.3.1 La mesure F

La précision de notre approche proposée a été évaluée en utilisant une mesure de validation externe. La mesure F (aussi appelée F-mesure), qui a été introduite pour la première fois par C. J. van Rijsberge dans [Rijsbergen, C.J.V. 1979], est un outil bien connu dans le domaine de la recherche d'informations (RI), et qui peut également être utilisé pour mesurer la qualité d'un partitionnement. La validation externe peut être utilisée lorsque le partitionnement réel des données est connu a priori. Ayant une idée sur les classes (ou catégories) des objets à partitionner, nous pouvons les comparer avec les clusters créés par un algorithme. La validation externe est connue pour être plus précise qu'une validation interne ou relative [Draszawka, K. and Szymański, J. 2011].

Nous supposons que pour un ensemble quelconque d'items à partitionner $O = \{o_1, o_2, \dots, o_n\}$, tel que $|O| = n$, nous avons à la fois: le vrai, correcte partitionnement de cet ensemble, dénoté $L = \{L_1, L_2, \dots, L_{K^L}\}$ (nous appelons les ensemble de $L$ *des classes*, tel que $K^L$ est le *nombre de classes*) et le partitionnement obtenu par un algorithme de partitionnement ou de classification, dénoté $C = \{C_1, C_2, \dots, C_{K^C}\}$ (les ensembles de $C$ sont des partitions, et $K^C$ est le nombre de partitions). Ayant une idée sur ces deux partitionnements $L$ et $C$, nous pouvons alors calculer leur similarité.

La mesure F est un mélange de deux indices: la *précision* (P), qui indique l'homogénéité des partitions par rapport aux classes prédéfinies, et le *rappel* (R), qui évalue la complétude des partitions relativement aux classes. Une précision élevée montre que presque tous les éléments de la partition correspondent à la classe attendue. Un rappel faible, d'autre part, indique qu'il existe plusieurs éléments réels qui n'ont pas été récupérés. La convenance de la mesure F par rapport à notre travail est justifiée par le comportement de cette mesure. En effet, la mesure F calcule la qualité de chaque partition indépendamment par rapport à chacune des classes prédéfinies, ce qui nous permet de déterminer automatiquement la partition la plus représentative pour chaque classe dans l'exemple de référence.

Ayant une idée sur la notation introduite précédemment, la précision d'une partition $C_i$ par rapport à une classe $L_j$ est calculée comme suit:

$$P(C_i, L_j) = \frac{|C_i \cap L_j|}{|C_i|} \quad (1)$$

Le rappel d'une partition $C_i$ par rapport à une classe $L_j$ est calculé comme suit:

$$R(C_i, L_j) = \frac{|C_i \cap L_j|}{|L_j|} \quad (2)$$

Ainsi, la valeur F d'une partition $C_i$ par rapport à une classe $L_j$ est la combinaison des deux:

$$F(C_i, L_j) = \frac{2 \times P(C_i, L_j) \times R(C_i, L_j)}{P(C_i, L_j) + R(C_i, L_j)} \quad (3)$$

Par conséquent, la valeur de F pour une partition $C_i$ est la plus élevée des valeurs de F obtenues en comparant cette partition avec chacune des classes prédéfinies:

$$F(C_i) = \max_{L_j \in L} F(C_i, L_j) \quad (4)$$

Enfin, la mesure F totale calculée pour tout le système de partitionnement est une moyenne pondérée des mesures F individuelles de toutes les partitions:

$$F_{totale} = \frac{1}{n} \sum_{C_i \in C} |L_j| \times \max_{L_j \in L} F(C_i, L_j) \quad (5)$$

Tel que n est le nombre total d'items à partitionner, et $|L_j|$ est le nombre d'items dans la classe $L_j$ qui correspond à la partition calculée $C_i$ (qui maximise sa valeur F).

#### 2.2.3.2 La mesure FBCubed

Malgré le fait que l'utilisation de la F-mesure diminue la participation de l'expert lors de l'étape d'évaluation, cette mesure semble être inappropriée lors de l'évaluation de l'*efficacité globale* d'une solution de partitionnement avec chevauchement. Selon Amigó et al. [Amigó, E. et al. 2009], F-mesure ne détectent pas toujours les petites améliorations dans le résultat de partitionnement, ce qui pourrait avoir des conséquences négatives dans les cycles d'évaluation et/ou de raffinement du système de partitionnement. Amigó et al. [Amigó, E. et al. 2009] ont proposé une nouvelle mesure, dénommé FBCubed, qui donne une bonne estimation de l'efficacité du système de partitionnement tout en considérant le chevauchement entre les partitions. Ainsi, nous avons décidé d'évaluer la précision et les performances globales de notre méthode proposée en utilisant FBCubed. La mesure FBCubed est calculée en utilisant la *Précision BCubed* et le *Rappel BCubed*, comme proposé dans [Amigó, E. et al. 2009]. La Précision BCubed et le Rappel BCubed sont basés sur la *Précision de Multiplicité* et le *Rappel de Multiplicité* respectivement; qui sont définis comme suit:

$$MP(o_1, o_2) = \frac{Min(|C(o_1) \cap C(o_2)|, |L(o_1) \cap L(o_2)|)}{|C(o_1) \cap C(o_2)|} \quad (6)$$

$$MR(o_1, o_2) = \frac{Min(|C(o_1) \cap C(o_2)|, |L(o_1) \cap L(o_2)|)}{|L(o_1) \cap L(o_2)|} \quad (7)$$

Tel que $o_1$ et $o_2$ sont deux items, $L(o_1)$ est l'ensemble de classes associées à $o_1$, $C(o_1)$ est l'ensemble de partitions associées à $o_1$. $MP(o_1, o_2)$ est la Précision de Multiplicité de $o_1$ par rapport à $o_2$, tel que $o_1$ et $o_2$ partagent au moins une partition. $MR(o_1, o_2)$ est le Rappel de Multiplicité de $o_1$ par rapport à $o_2$, tel que $o_1$ et $o_2$ partagent au moins une classe.

Soit $D(o_i)$ l'ensemble d'items qui partagent au moins une partition avec l'item $o_i$, y compris $o_i$ lui-même. La Précision BCubed de l'item $o_i$ est définie comme suit:

$$BCubed_{Precision}(o_i) = \frac{\sum_{o_j \in D(o_i)} MP(o_i, o_j)}{|D(o_i)|} \quad (8)$$

Soit $H(o_i)$ l'ensemble d'items qui partagent au moins une classe avec l'item $o_i$, y compris $o_i$ lui-même. Le Rappel BCubed de l'item $o_i$ est défini comme suit:

$$BCubed_{Rappel}(o_i) = \frac{\sum_{o_j \in H(o_i)} MR(o_i, o_j)}{|H(o_i)|} \quad (9)$$

La Précision BCubed *globale* d'une solution de partitionnement, notée $BCubed_{Precision}$, est la moyenne des Précisions BCubed de tous les items dans la distribution $O$; le Rappel BCubed d'une solution de partitionnement, noté $BCubed_{Rappel}$, est défini d'une manière analogue mais en utilisant le Rappel BCubed de tous les items. Finalement, la mesure BCubed de la solution de partitionnement est la moyenne harmonique de $BCubed_{Precision}$ et $BCubed_{Rappel}$ comme suit:

$$FBCubed = \frac{2 \times BCubed_{Precision} \times BCubed_{Rappel}}{BCubed_{Precision} + BCubed_{Rappel}} \quad (10)$$

## 2.3 Les ontologies et la recherche d'information

Les ontologies sont un moyen pour représenter la connaissance. Ces représentations de connaissances correspondent à «une spécification explicite et formelle d'une conceptualisation partagée» [Studer, R. et al. 1998]. Une ontologie est donc un moyen pour spécifier les relations entre les concepts, les objets et les autres entités appartenant à un domaine particulier. Les ontologies sont utilisées pour résoudre des problèmes dans un large éventail d'applications et de domaines, parmi lesquelles on trouve le domaine de la recherche d'information (RI). La recherche d'information consiste à trouver du matériel (i.e. documents) d'une nature non structurée (i.e. texte) qui satisfait un besoin en information, à partir de grandes collections [Manning, C.D. et al. 2008]. L'objectif principal d'un système de RI est donc de minimiser les ressources humaines nécessaires pour trouver les informations nécessaires afin d'accomplir une tâche [Kowalski, G.J. and Maybury,

M.T. 2002]. Afin de pouvoir satisfaire les besoins de l'utilisateur en information, un système de RI doit donc capturer les sens transmis dans les granules documentaires (ou autres ressources d'information), ainsi que ceux dans le besoin (i.e. requête) de l'utilisateur. Une fois le système de RI capture les conceptualisations associées avec le contenu et le besoin, il pourra donc établir des correspondances entre les deux. C'est à cette étape là qu'intervient l'ontologie en jouant le rôle d'une couche sémantique afin de pouvoir comparer des choses.

Il existe plusieurs techniques à base d'ontologies telles que la classification (aussi appelée *la catégorisation*) des documents, la classification de gènes dans le domaine de la bio-informatique, … etc. L'une de ces techniques, du quelle nous nous sommes inspirés, est la classification non-supervisé des documents. La classification des documents (i.e. *documents clustering*) est un sous domaine de la recherche d'information qui vise à organiser automatiquement les documents textuels dans des groupes significatifs, de telle sorte que tous les documents du même groupe ont une grande similarité (intragroupe) et une dissemblance entre les groupes (intergroupes). Plusieurs approches ont été proposées dans la littérature pour mieux classifier les documents textuels. Indépendamment de la technique utilisée, lorsque cette dernière est appliquée à la classification, quatre questions sont à considérer [Sureka, V. and Punitha, S.C. 2012]:

1. La sélection des éléments ;
2. La dimensionnalité de l'espace d'éléments ;
3. Le processus de classification (calcul de similarité);
4. L'algorithme utilisé pour la classification.

La sélection des éléments importants, aussi appelée *l'indexation* des granules documentaires, est le processus d'identification de descripteurs représentatifs de leurs contenus, i.e. des termes de qualité, qui ont un impact positif sur la performance de la classification, en supprimant les données redondantes ou impertinentes. Les descripteurs ainsi sélectionnés sont normalement d'une grande dimension, ce qui impose un grand défi pour la performance des algorithmes de classification. Il existe plusieurs types d'indexation allant des approches classiques reposant sur une analyse lexicale jusqu'aux approches sémantique reposant sur l'utilisation d'ontologies modélisant la conceptualisation des objets cités dans le corpus [Hernandez, N. 2005].

Clairement, la notion de « *similarité* » entre les documents doit être formalisée. Cela est fait en définissant une distance entre tout couple de documents. Pour ce faire, la plupart des approches (dites classiques) de classification de documents se basent sur un modèle VSM (pour *Vector Space Model*). Ce type de modèle a été utilisé pour modéliser le contenu d'un document selon une approche algébrique. Néanmoins, cette technique ne considère pas la sémantique des descripteurs

(termes) sélectionnés. En effet, les descripteurs des documents générés par les approches classiques sont souvent exposés aux plusieurs problèmes, dont les plus connus sont celui de synonymie et de polysémie. Pour résoudre ce problème, des techniques de classification à base d'ontologies sont couramment utilisées dans les dernières décennies [Mabotuwana, T. et al. 2013]. Une approche basée sur l'ontologie permet d'exploiter les informations sur les relations sémantiques entre concepts de l'ontologie. Une méthode d'indexation à base de concept considère les mots-clés dans le document, ainsi que les concepts basés sur la connaissance du domaine. Un document peut contenir plusieurs concepts. Après l'extraction des concepts à partir du document, des *poids sémantiques* sont calculés pour effectuer une indexation et, par conséquent, une catégorisation efficace des documents.

### 2.3.1 La synergie entre les MF et les ontologies

Il existe une forte ressemblance entre l'ingénierie des ontologies du web sémantique et la modélisation de fonctionnalités. En effet, ils représentent tous deux les concepts dans un domaine particulier et définissent la manière dont les différentes propriétés sont liées entre elles. Donc, Si on arrive à établir une correspondance entre un MF et une ontologie, cette dernière peut donc améliorer l'utilisation du MF par les moyens suivants:

- Fournir les vocabulaires et les termes précis pour les ingénieurs des LdP, ce qui permet d'augmenter le niveau de standardisation et d'interopérabilité, et d'éviter l'ambiguïté et les duplications ; lors de l'extraction des fonctionnalités, on peut avoir beaucoup de problèmes dus aux mauvaises compréhensions et aux duplications respectivement. Par exemple, le cas de deux fonctionnalités ayants le même nom avec deux comportements différents, ou bien deux fonctionnalités ayants le même comportement et qui sont représentées par deux notations différentes.
- Vérifier la cohérence entre un modèle de fonctionnalités et une ontologie, par exemple, les contraintes exprimées dans l'ontologie sont-elles exprimées aussi dans le MF ?
- La restructuration de la hiérarchie d'un MF et l'organisation des relations entre ses fonctionnalités. Cela peut être très utile dans une approche extractive des LdP dont l'enjeu majeur pour construire une hiérarchie est d'identifier, pour chaque fonctionnalité dans cette hiérarchie, la fonctionnalité ascendante (i.e. fonctionnalité mère) [She, S. et al. 2011].
- Définir la sémantique d'un MF en inférant automatiquement la sémantique des fonctionnalités et leurs relations.
- Automatiser la dérivation et la validation de produits à partir d'une LdP en profitant des moteurs de raisonnement sur les ontologies déjà disponibles.

- Les correspondances établies entre une ontologie et les parties de programmes (les implémentations des fonctionnalités) permettent aux développeurs de considérer ces parties du point de vue de concepts qu'ils mettent en œuvre.
- La plupart des approches actuelles de rétro-ingénierie de MF à partir du code source sont à un niveau purement lexical. Toutefois, l'interprétation de leurs résultats nécessite encore des informations sémantiques en prenant en considération les connaissances du monde réel.

Johansen et al. [Johansen, M.F. et al. 2010] ont déjà exploré la synergie entre les deux formalismes, et ont proposé quelques stratégie ainsi que quelques règles pour établir la correspondance entre un MF et une ontologie. Néanmoins, les deux formalismes se diffèrent dans certains points, ce qui rend cette tâche difficile:

- **L'hypothèse du monde ouvert et du monde fermé**[2] : contrairement aux MF, les ontologies sont utilisées pour représenter des connaissances incomplète (utilise l'hypothèse du monde ouvert *OWA*). Donc, une arête entre deux fonctionnalités dans un MF ne correspond pas nécessairement à une relation de l'ontologie.
- **Le but du model :** un MF modélise quelque chose de spécifique (LdP) tandis qu'une ontologie modélise quelque chose plus générale (domaine). Donc, on doit prendre ce point en considération lors la vérification de la consistance entre un MF et une ontologie.
- **La granularité :** même si une ontologie et un MF modélise la même chose, ils peuvent le faire à différent niveaux de granularité.
- **Le focus :** même si une classe et une fonctionnalité sont du même niveau de granularité, elles peuvent ne pas avoir le même focus (ex. un téléphone portable peut être considéré comme un appareil photo dans un modèle et un lecteur mp3 dans un autre).
- **Chevauchement de l'information :** Dans la relation entre les deux modèles, on ne veut relier que les parties pertinentes qui parlent des mêmes choses. L'identification des parties pertinentes ne peut pas être entièrement automatisée et requiert la connaissance des utilisateurs.
- **Les cardinalités :** Ceci est dû au fait que les MF ont une cardinalité de type 0…1 pour chaque fonctionnalité, tandis qu'une ontologie nous permet d'avoir cardinalité 0...N.

[2] *Open World Assumption (OWA)* suppose que ce qui n'est pas connu pour être vrai, c'est tout simplement inconnu. *Closed World Assumption (CWA)* suppose que ce qui n'est pas connu pour être vrai doit être faut.

Les problèmes discutés ci-dessus semblent être tous liés, directement ou indirectement, à l'hypothèse du monde ouvert OWA. Donc, trouver une solution pour ce dernier, en fermant (restreignant) les classes, est susceptible d'aider à surmonter ces difficultés. Néanmoins, cette solution est actuellement peu fréquente.

### 2.3.2 L'indexation à partir d'ontologies

L'indexation (initialement appelée Catalogage) au sens large est définie par Kowalski comme « la technique la plus ancienne pour identifier le contenu des éléments afin de faciliter leur extraction. L'objectif du catalogage est de donner des points d'accès à une collection, qui sont espérés et plus utiles aux utilisateurs de l'information » [Kowalski, G.J. and Maybury, M.T. 2002]. Un exemple représentatif de l'indexation est celui d'indexation de documents. Pour former des points d'accès qui jouent le rôle de descripteurs pour un document, les techniques classiques d'indexation de documents avaient tendance à utiliser comme langage de représentation de documents des mots-clés extraits à partir de ces documents sujets d'indexation ou bien d'utiliser un vocabulaire contrôlé. Ce faisant, les seuls documents récupérés lors d'une recherche lancée par l'utilisateur sont ceux contenant des mots-clés (i.e. descripteurs) qui correspondent à la requête de ce dernier. Les techniques classiques d'indexation utilisent généralement le modèle VSM (pour Vector Space Model) pour modéliser le contenu d'un document sous forme d'un vecteur d'identificateurs. Chaque élément du vecteur correspond à un terme séparé. Par exemple, si un terme existe au moins une fois dans le document, sa valeur dans le vecteur est donc différente de zéro. On appels cette valeur le «poids » du terme. L'un des schémas les plus connus pour calculer de telles pondérations de termes est le schéma statistique TF-IDF (pour Term Frequency-Inverse Document Frequency) [Salton, G. and Buckley, C. 1988].

Les techniques classiques d'indexation souffrent de certaines lacunes et les résultats retournés suite à une requête d'utilisateur sur des documents qui sont indexés utilisant ces techniques sont souvent remplis de bruit et contiennent des documents impertinents, ce qui nécessite un effort supplémentaire de la part de l'utilisateur pour filtrer ces résultats et sélectionner les documents désirés. De plus, beaucoup de documents peuvent transmettre l'information sémantique désirée par l'utilisateur sans pour autant contenir ces mots-clés.

L'indexation à base d'ontologies, aussi appelée l'indexation sémantique, a été donc proposée pour surmonter ce problème. Le principe de base dans un processus d'indexation sémantique est d'indexer les granules documentaires utilisant les sens de termes, identifiés à partir d'une ontologie, plutôt que leurs représentations lexicales [Mihalcea, R. and Moldovan, D. 2000]. Le sens des informations transmises par le texte d'un document dépend, donc, fortement des liens sémantiques entre

les concepts ontologiques auxquels le texte se réfère. L'ontologie représente ainsi une pièce angulaire dans un processus d'indexation sémantique. Selon la catégorie de l'ontologie utilisée, certaine auteurs distingue deux classes d'indexations [Mihalcea, R. and Moldovan, D. 2000]: (1) une indexation conceptuelle utilisant une ontologie de domaine, et (2) une indexation sémantique utilisant une ontologie terminologique telle que WordNet.

Quel que soit le type d'ontologie à utiliser le processus d'indexation sémantique doit passer, comme dans le cas des techniques classiques d'indexation, par deux tâches/problèmes distincts: (1) l'identification des concepts (resp. leurs instances) de l'ontologie dans les granules documentaires, et (2) le calcul de leurs poids relatifs.

#### 2.3.2.1 Identification des concepts et des instances de l'ontologie

L'identification des concepts (resp. leurs instances) consiste à relier un concept dans l'ontologie à un terme (resp. une expression) dans les granules documentaires qui traduit le mieux le sens (la sémantique) incorporé dans ce concept. Nous distinguons deux types de procédés pour réussir une telle tâche : (1) manuel et (2) automatique.

Les approches manuelles utilisées dans [Khan, L. and Luo, F. 2002, Paralic, J. and Kostial, I. 2003, Vallet, D. et al. 2005] reposent sur le travail d'un expert pour établir les correspondances entre les termes documentaires et les concepts de l'ontologie. Néanmoins, ce procédé est couteux en termes d'effort et de temps et reste exposé aux erreurs, étant donné qu'il dépend fortement de la compétence et de la fiabilité de l'expert en question.

D'autres approches visent à automatiser ce procédé. Les labels ou termes désignant les concepts ou instances sont recherchés automatiquement dans les granules documentaires utilisant des techniques d'indexation classique (la tokenisation, l'élimination des mots vides, et la lemmatisation). Un concept (resp. instance) est en effet défini à partir d'un ou plusieurs labels dans le document consistant à des variantes lexicales du terme qui définit le concept. Cependant, relier un terme spécifique à un concept dont ce terme est une variante lexicale est un processus complexe car un terme peut représenter plusieurs concepts différents à des degrés différents [Kowalski, G.J. and Maybury, M.T. 2002]. Par exemple, le terme «automobile» pourrait être associé à des concepts tels que «véhicule», «transport», «dispositif mécanique», «carburant» et «environnement». Le terme «automobile» est fortement liée à «véhicule», moindrement à «transport» et beaucoup moins aux autres concepts. Pour remédier à ce problème, un mécanisme de désambiguïsation des sens des termes doit être utilisé pour relier le terme à son concept exact.

Plusieurs approches de désambiguïsation ont été proposées dans la littérature dont les plus simples utilisent les trois techniques suivantes [Hernandez, N. 2005] :

1. La stratégie du *« tout »* consiste à considérer tous les concepts dont le terme est une instance;
2. La technique du *« premier »* consiste à choisir le concept le plus fréquent dans le document (ou dans la collection) ;
3. La technique du *« contexte »* consiste à utiliser les termes qui apparaissent dans le même contexte du document avec les concepts candidat, et de s'appuyer sur la proximité sémantique des concepts pour identifier le sens exact.

#### 2.3.2.2 Pondération des concepts et instances

La pondération est un problème crucial dans la RI. En effet, la qualité des résultats récupérés dépend fortement de la qualité de la pondération. Alors que les approches classiques d'indexation utilisent des approches purement statistiques de pondération telle que le schéma TF-IDF, le défi dans le contexte d'indexation conceptuelle est de savoir comment pondérer correctement les concepts.

A ce propos, plusieurs approches ont été proposées dans la littérature pour pondérer les concepts dans le contexte d'indexation sémantique, et qui peuvent être classées en deux catégories [Boubekeur, F. et al. 2010] : (1) les techniques statistiques de pondération de concepts, et (2) les techniques sémantiques de pondération de concepts.

Dans les approches statistiques de pondération [Baziz, M. et al. 2005, Vallet, D. et al. 2005], on s'inspire de la méthode TF-IDF. Les concepts ontologiques sont désignés à travers les termes qui les représentent dans un document. La pondération des concepts se résume, par conséquent, en une pondération classique en calculant la fréquence d'apparition des concepts. L'inconvénient de cette approche est qu'elle ignore l'organisation conceptuelle des concepts repérés dans le document et, par conséquent, la sémantique contenue dans les relations entre ces concepts.

En revanche, les approches de pondération sémantique de concepts visent à évaluer la représentativité d'un concept ontologique désigné dans le contenu d'un document en se basant, en plus de sa fréquence d'apparition, sur le nombre de liens ainsi que la similarité sémantique de ce concept avec d'autres concepts du document. Le calcul de la similarité sémantique entre les concepts est basé sur leur positionnement dans l'ontologie (voir section 2.3.3). Un exemple illustratif de cette technique est proposé par Desmontils [Desmontils, E. and Jacquin, C. 2002]. Dans son approche, Desmontils utilise deux étapes pour extraire les concepts représentatifs d'une page web. Dans un premier temps on construit un index des

termes représentatifs de la page, dont chaque terme de cette liste est associé à sa fréquence. Cette fréquence dépend du nombre d'apparition ainsi que le poids de la balise HTML qui entoure chaque occurrence du terme. Dans une page web contenant *n* termes différents, et pour un terme donné $T_i$ ($i \in [1,n]$) qui apparait $p$ fois dans la page, la fréquence pondérée $P\_Freq(T_i)$ est calculée comme la somme des poids des balises HTML associées aux occurrences du terme $T_i$. Cette fréquence est donnée comme suit :

$$P_{Freq(T_i)} = \frac{P(T_i)}{max_{k=1..n}\big(P(T_k)\big)}; \quad P(T_i) = \sum_{i=1}^{p} M_{i,j} \tag{11}$$

Où $M_{i,j}$ correspond au poids de la balise HTML relative à la $j^{éme}$ occurrence du terme $T_i$.

Dans une deuxième étape, une ontologie (WordNet) est ensuite utilisée pour générer tous les concepts candidats (i.e. les sens) qui peuvent être étiqueté par un terme de l'index précédent. La représentativité du contenu de la page web pour un concept candidat est évaluée en se basant sur sa fréquence ainsi que ses relations avec d'autres concepts de la même page. Cela permet de choisir le meilleur sens (concept) d'un terme par rapport au contexte (i.e. une désambiguïsation à base du contexte). Par conséquent, plus un concept possède de fortes relations avec d'autres concepts de sa page, plus ce concept est significatif dans sa page. Cette relation contextuelle minimise le rôle de la fréquence pondérée en augmentant le poids des concepts fortement liés et en affaiblissant les concepts isolés (même avec une forte fréquence pondérée). Un sens (concepts candidat) est représenté par une liste de synonymes (i.e. synset). Pour chaque concept candidat, la représentativité est calculée en fonction de la fréquence pondérée et de la similarité cumulée du concept avec les autres concepts de la page. La mesure de similarité cumulative relative à un concept dans une page, noté $\widehat{sim}$ , est la somme de toutes les mesures de similarité calculées entre ce concept et tous les autres concepts inclus dans la page étudiée. Pour calculer la similarité $sim$ entre deux synset, Desmontils utilise la mesure de Wu & Palmer [Wu, Z. and Palmer, M. 1994]. La mesure de similarité cumulative $\widehat{sim}$ est donnée comme suit :

$$\widehat{sim}\big(synset_i(T_k)\big) = \sum_{j\in[1,k-1]\cup[k+1,m]} \sum_{l=1}^{l_j} sim\Big(synset_i(T_k), synset_l\big(T_j\big)\Big) \tag{12}$$

Où $l_k$ synsets sont associés à un terme $T_j$, et il y a $m$ termes dans les pages web étudiées. Enfin, on calcule un coefficient de représentativité qui détermine la représentativité d'un concept dans un document. Le coefficient est une combinaison linéaire de la fréquence pondérée et de la similarité cumulative d'un concept :

$$Rep\left(synset_i(T_k)\right) = \frac{\alpha * P_{Freq(synset_i(T_k))} + \beta * \widehat{sim}\left(synset_i(T_k)\right)}{\alpha + \beta} \qquad (13)$$

Tel que $\alpha$ et $\beta$ sont fixé expérimentalement à 1 et 2 [Desmontils, E. and Jacquin, C. 2002]. Dans ce calcul, toutes les similarités ne sont pas prises en compte pour discriminer les résultats: un seuil est appliqué.

### 2.3.3 La similarité sémantique

La similarité sémantique est une évaluation du lien sémantique entre deux concepts dont le but est d'estimer le degré par lequel les concepts sont proches dans leur sens [Resnik, P. 1999]. La notion de similarité sémantique (*semantic similarity*) est différente de celle de la proximité sémantique (*semantic relatedness*) [Harispe, S. et al. 2013]. En effet, la première est incluse dans la deuxième, mais les deux termes peuvent être utilisés de manière interchangeable dans certains contextes, ce qui rend encore plus important d'être conscients de leur distinction. Deux concepts sont considérés comme sémantiquement similaires s'il existe une relation sémantique lexicale entre eux, par exemple, une synonymie (ou quasi-synonymie), hyponymie, antonymie, ou relation de troponymy entre eux. Deux sens de mots sont considérés comme sémantiquement liés s'il n'existe aucune relation sémantique lexicale entre eux ; ce sont des éléments qui apparaissent souvent ensemble comme, par exemples, pommes-bananes, chirurgien-scalpel, arbre-ombre. La majorité des travaux portant sur le calcul de similarité dans une ontologie considèrent que la similarité peut être évaluée uniquement à partir des liens taxonomiques (i.e. «*est-un*») [Rada, R. et al. 1989, Wu, Z. and Palmer, M. 1994, Resnik, P. 1995, Jiang, J.J. and Conrath, D.W. 1997].

#### 2.3.3.1 WordNet

WordNet [Miller, G.A. 1995] est le produit d'un projet de recherche à l'Université de Princeton. WordNet est une grande base de données lexicale de la langue anglaise. Dans WordNet les verbes, les noms et les adjectifs sont organisés dans des ensembles de synonymes. Un synset est un ensemble de mots reliés par une relation symétrique de synonymie, dénotant un concept, et qui sont interchangeable dans plusieurs contextes [Miller, G.A. 1995]. Les synset sont reliés utilisant plusieurs types de relations sémantiques.

Hormis la synonymie utilisée pour former les synset, des exemples de relations sémantiques utilisées par WordNet sont l'antonymie, hyponymie, membre, …etc. Ces relations sont associées avec des mots pour former une structure hiérarchique, ce qui en fait un outil utile pour le traitement automatique du langage naturel. En effet, il est communément affirmé que la sémantique d'un langage est surtout capturée par des noms ou des phrases nominales de telle sorte que la plupart des recherches se concentrent sur les noms dans le calcul de similarité sémantique. Il

existe quatre types de relations sémantiques couramment utilisées pour les noms, qui sont : hyponyme/hyperonyme (est-un), partie-méronyme/partie-holonyme (partie-de), membre-méronyme/membre-holonyme (membre-de) et la substance-méronyme/substance-holonyme (substance-de). Par exemple, la pomme est un fruit (est-un) et le clavier fait partie de l'ordinateur (partie-de). La relation Hyponyme/hyperonyme « est-un » est la relation la plus commune, qui représente près de 80% des relations dans WordNet. Un fragment de la relation « *est-un* » entre les concepts dans WordNet version 3.0 est représenté dans figure *Fig 2.6*. Dans la taxonomie, les concepts plus profonds sont plus spécifiques et le concept supérieur sont plus abstrait. Par conséquent « *Digital_computer* » est plus abstrait que « *File_server* », et « *Data_processor* » est plus abstrait que « *Digital_computer* ». Le concept « *Entity* » est le concept le plus abstrait.

Les adjectifs quant à eux sont organisés en termes d'antonymie. Les paires d'antonymes « *directs* » comme *humide-sec* et *jeune-vieux* reflètent un fort contrat sémantique de leurs membres. Chacun de ces adjectifs polaires, à son tour, est lié à un certain nombre d'adjectifs qui sont "sémantiquement similaires": *sèche* est liée à *aride*, *desséchée* et *sec*, et *humide* à *détrempé*, *… etc*. Les adjectifs qui sont sémantiquement similaire à « *sèche* » sont des *antonymes indirects* du mot « *humide* » ainsi que les adjectifs qui lui sont similaires.

En général, le résultat obtenu par rapport à la relation « *est-un* » est considéré comme étant la similarité entre les concepts. Le résultat obtenu à partir d'autres relations, telles que « *partie-de* » est considéré comme une proximité sémantique. Dans notre travail, nous sommes seulement préoccupés par le calcul de similarité entre les noms et la relation hyponyme/hyperonyme de WordNet.

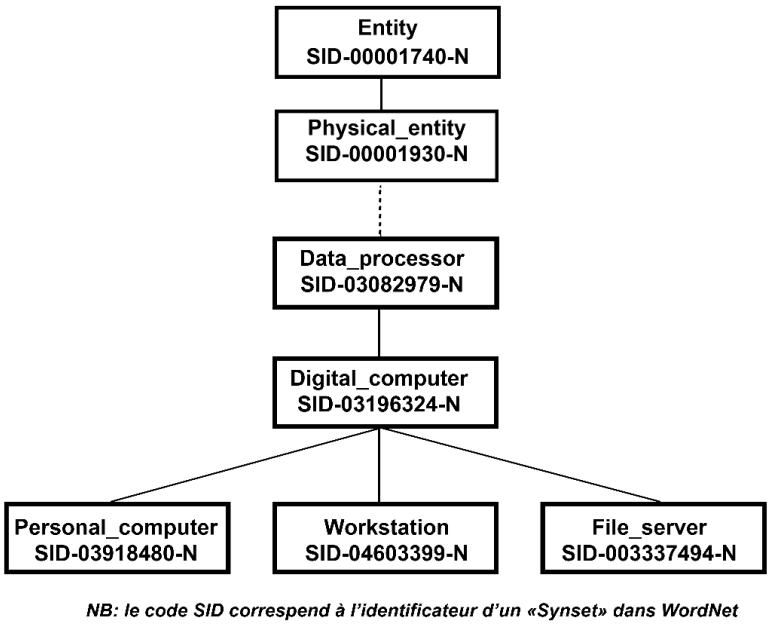



**Fig 2.6 : Un fragment de la relation « *est-un* » dans WordNet 3.0.**

#### 2.3.3.2 Mesures de similarité à base de WordNet

Différentes mesures ont été définies pour permettre le calcul de la similarité entre les concepts. On peut distinguer deux catégories principales de mesures[Mohammad, S.M. and Hirst, G. 2012] : (01) les mesures basées sur la richesse, en terme de connaissances, des ressources utilisées telles que les ontologies (ex. WordNet) ou les ressources terminologiques (dictionnaires, thésaurus, …etc), et (02) celles basées sur le corpus ou ce qu'on appelle les *mesures distributionnelles*. Nous nous intéressons dans le cadre de notre travail au premier type de mesures, i.e. celles basées sur des ontologies. La création d'ontologies et de réseaux sémantiques tel que WordNet a permis leur utilisation pour aider à résoudre de nombreux problèmes de langage naturel, y compris la mesure de la distance sémantique. Les mesures basées sur les ontologies sont classées par rapport aux caractéristiques des concepts permettant d'évaluer la similarité. Ces caractéristiques reposent soit sur la distance entre deux concepts à travers leurs liens dans l'ontologie, soit sur l'information contenue par les concepts, soit sur les deux [Slimani, T. et al. 2007, Mohammad, S.M. and Hirst, G. 2012, Tchechmedjiev, A. 2012, Anitha Elavarasi, S. et al. 2014].

L'idée de base dans une approche structurelle à base de distance, est de quantifier la similarité entre deux concepts en se basant sur le chemin qui les relie dans la hiérarchie (ontologie). La technique de comptage des arêtes (Edge Counting) [Rada, R. et al. 1989] est la première technique qui a été proposée dans ce sens. D'autres variantes ont été ensuite proposées, comme la mesure de Wu & Palmer [Wu, Z. and Palmer, M. 1994] qui combine la longueur de chemin avec la notion de profondeur de concepts. Les mesures reposant sur la distance souffrent de certains défauts. En effet, les liens dans une ontologie ne reflètent toujours pas la distance sémantique. De plus, les choix arbitraires qui sont faits durant la phase de conception de l'ontologie sont susceptibles de falsifier les résultats de similarité. Des approches basées sur le contenu informationnel ont été donc proposées pour remédier à ce problème.

Les approches basées sur le contenu informationnel [Resnik, P. 1995, Jiang, J.J. and Conrath, D.W. 1997, Lin, D. 1998] supposent que l'information détenue par les concepts puisse être quantifiée. La similarité est alors calculée à partir de l'information partagée par les concepts. Ces approches adoptent une nouvelle mesure en termes de la mesure d'entropie issue de la théorie de l'information. Afin de calculer l'information contenue par les concepts, un corpus de référence est choisi. Les concepts sont alors pondérés par une fonction correspondant à l'information portée par un concept dans le corpus ; on associe des probabilités à chaque concept dans la taxonomie (i.e. wordNet) en se basant sur le nombre d'occurrences de mots dans le corpus de référence. Ces probabilités sont cumulatives vu qu'on parcourt la taxonomie allant des concepts spécifiques à des

concepts plus abstraits. Cela signifie que toutes les occurrences d'un nom dans le corpus sont également considérées comme des occurrences de chacune des classes taxonomiques qui le subsument. Resnik [Resnik, P. 1995] a été le premier à considérer l'utilisation de cette formule, qui provient des travaux de Shannon [Shannon, C.E. 1948], pour le calcul de la similarité sémantique. Le contenu en information (CI) d'un concept $c$ est calculé de la façon suivante [Resnik, P. 1995]:

$$CI_{Resnik}(c) = -\log p(c) \tag{14}$$

Où $c$ est un concept donné dans WordNet et $p(c)$ est la probabilité de tomber sur une occurrence de $c$ dans un corpus de référence. Cette méthode garantit que la valeur CI décroit d'une façon monotone allant des feuilles de la taxonomie jusqu'à ses racines. L'intuition de base derrière l'utilisation de la probabilité négative est que le plus probable qu'un concept est susceptible d'apparaitre le moins d'informations qu'il véhicule, en d'autres termes, les mots rares sont plus informatifs que les mots fréquents. Ayant idée sur les valeurs CI pour chaque concept, nous pouvons alors calculer la similarité sémantique entre deux concepts donnés. Selon Resnik, la similarité sémantique dépend de la quantité d'informations que deux concepts $c_1$ et $c_2$ ont en commun. Cette information partagée est basée sur la détermination du contenu informationnel du concept commun le plus spécifique (notée $lso(\mathrm{c}_1, c_2)$ pour *lowest superordinate*) qui subsume $c_1$ et $c_2$. Si $lso(\mathrm{c}_1, c_2)$ n'existe pas les deux concepts sont alors complétement dissemblables, sinon l'information partagée est égale à la valeur CI de $lso(\mathrm{c}_1, c_2)$. Formellement, la similarité sémantique de Resnik est définie comme suit:

$$\mathrm{Sim}_{Resnik}(c_1, c_2) = CI\big(lso(c_1, c_2)\big) = \mathrm{Max}_{c \in S(c_1, c_2)}\big[-\log\big(p(c)\big)\big] \tag{15}$$

$$Freq\big(lso(\mathrm{c}_1, c_2)\big) = \sum_{n \in word\left(lso(\mathrm{c}_1, c_2)\right)} count(n) \tag{16}$$

$$P\big(lso(\mathrm{c}_1, c_2)\big) = \frac{Freq\big(lso(\mathrm{c}_1, c_2)\big)}{N} \tag{17}$$

Où $S(c_1, c_2)$ est l'ensemble de concepts qui subsument à la fois $c_1$ et $c_2$. Ainsi, $c$ représente le concept le plus spécifique ayant la plus faible probabilité (i.e. qui maximise la valeur de similarité) qui subsume les deux concepts $c_1$ et $c_2$ dans l'ontologie, $word(lso(c_1, c_2))$ est l'ensemble des termes ou labels représentant le concept $lso(c_1, c_2)$ et les concepts subsumés par $lso(c_1, c_2)$, $P$ est la probabilité de trouver une instance du concept $lso(c_1, c_2)$, $N$ est le nombre total des instances (labels) de concepts retrouvés dans le corpus. Quand l'ontologie permet l'héritage multiple et que plusieurs concepts subsument les deux concepts considérés, cette formule permet de choisir dans l'ensemble des concepts candidats le concept le plus spécifique au sens où c'est celui qui a la probabilité la plus faible.

D'autres mesures basées sur celle de Resnik ont été proposées plus tard par d'autres chercheurs [Jiang, J.J. and Conrath, D.W. 1997, Lin, D. 1998]. Lin définit la

similarité sémantique comme suit : « la similarité sémantique entre A et B est quantifiée par le ratio entre la quantité d'information nécessaire pour indiquer les points commun entre A et B et l'information nécessaire pour décrire complètement ce que A et B sont ». Ainsi, en plus de la quantité d'information partagée par deux concepts proposée par Resnik, la mesure de Lin prend en considération aussi la différence entre ces deux concepts. La mesure de Lin vise à remédier à une limitation dans la mesure de Resnik qui attribue la même valeur de similarité à deux couples de concepts qui ont le même concept commun ($lso$).

$$Sim_{Lin}(c_1, c_2) = \frac{2 * \mathrm{Sim}_{Resnik}(c_1, c_2)}{CI_{Resnik}(c_1) + CI_{Resnik}(c_2)} \quad (18)$$

Jiang [Jiang, J.J. and Conrath, D.W. 1997] a proposé une mesure similaire à celle de Lin et qui combine la notion de nombre de liens avec celle du contenu informationnel. La différence avec la mesure de Lin et que la mesure de Jiang permet de calculer la distance sémantique au lieu de calculer la similarité. La mesure de similarité proposée par Lin et la mesure de distance de Jiang ont été proposées pour corriger des limitations dans la formule de Resnik qui ne permet pas d'obtenir des valeurs entre 0 et 1, mais plutôt une valeur donnée par $CI_{Resnik}(lso)$, tel que $CI_{Resnik} \in [0, \infty[$. Ce problème a été corrigé de sorte que $Sim_{Lin}(c_1, c_1) = 1$ et $Distance_{Jiang}(c_1, c_1) = 0$.

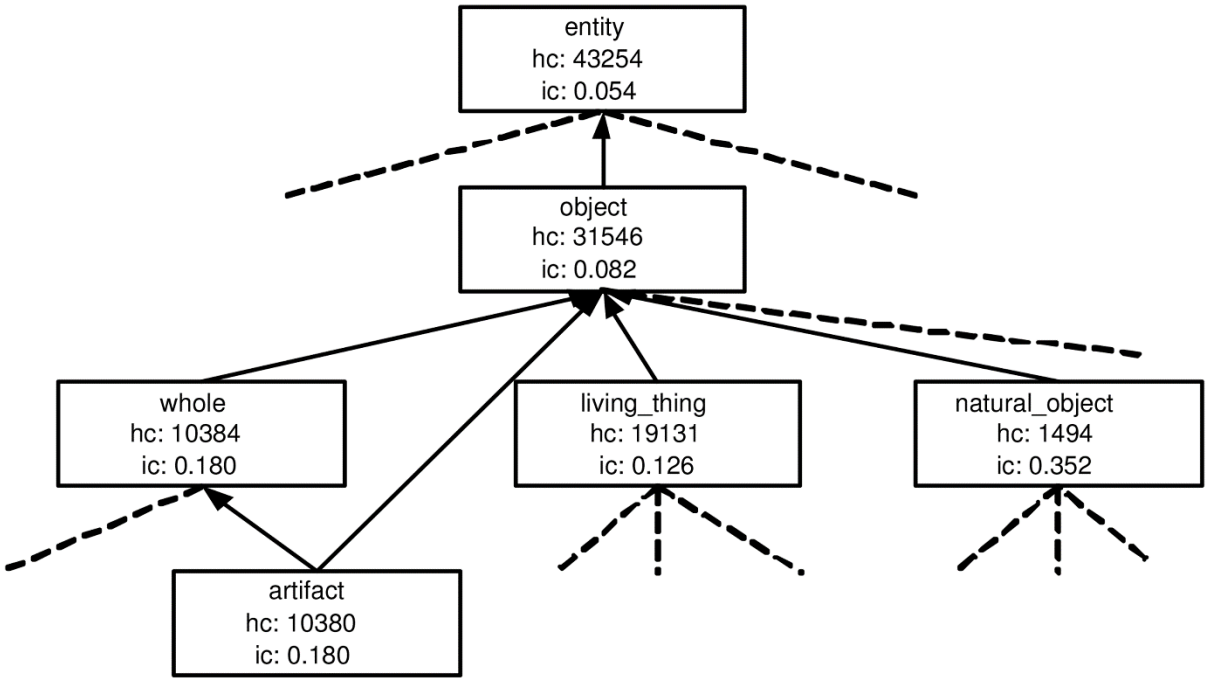
Seco et al. [Seco, N. et al. 2004] ont proposé une nouvelle mesure de similarité qui est complètement dérivée de WordNet sans avoir besoin de ressources externes (i.e. corpus) à partir desquelles les données statistiques sont recueillies. Ils expriment la valeur CI d'un concept dans WordNet en fonction des hyponymes qu'il possède. Ils supposent que la structure taxonomique de WordNet est organisée d'une manière significative et structurée, où les concepts avec de nombreux hyponymes transmettent moins d'informations que les concepts qui sont des feuilles; la valeur CI d'un concept décroit en augmentant le nombre d'hyponymes relatifs (i.e. il devient plus abstrait), et augmente en allant plus profondément dans la structure, et se maximise pour les concepts feuilles. Ces derniers sont considérés comme les plus spécifiques et portent une quantité importante d'information.

Les liens sémantiques d'hyperonymie/hyponymie sont les plus dominants parmi les relations inter-concepts de WordNet (approximativement 80%). D'où, on trouve que la majorité des approches proposées dans la littérature ont essayé donc de tirer profit de cette dernière caractéristique, et même considèrent cette relation comme suffisante pour calculer la similarité sémantique [Rada, R. et al. 1989]. La formule de Seco pour calculer la valeur CI pour un concept est donnée comme suit [Seco, N. et al. 2004]:

$$CI_{seco}\,(c) = 1 - \log \frac{(hypo(c) + 1)}{\log(max_{\mathrm{wn}})} \quad (19)$$

Tel que $hypo(c)$ est le nombre d'hyponymes du concept $c$, et $max_{\text{wn}}$ est une constante qui représente le nombre total de concepts de l'ontologie WordNet. La formulation ci-dessus garantit que le contenu en information diminue d'une façon monotone allant des nœuds feuilles aux nœuds racines comme illustré dans la figure *Fig 2.7*. De plus, le contenu en information du nœud supérieur imaginaire de WordNet donnerait une valeur $CI_{seco} = 0$.

Comme résultat de l'héritage multiple dans certains des concepts de WordNet, des précautions doivent être prises de telle sorte que chaque hyponyme distinct n'est considéré qu'une seule fois. Considérons à nouveau la situation dans la figure *Fig 2.7*, le concept « *artifact »* est un hyponyme direct des concepts « *whole »* et « *object ».* Étant donné que « *whole »* est aussi un hyponyme de « *object »*, nous ne devons pas considérer les hyponymes du concept « *artefact »* deux fois lors du calcul du nombre d'hyponymes de « *object »*.




***NB : ic et hc représentent respectivement le contenu en information et le nombre d'hyponymes.***

**Fig 2.7 : Un exemple de l'héritage multiple dans WordNet [Seco, N. et al. 2004].**

On peut remplacer $CI$ par $CI_{seco}$ dans toutes les formules précédentes pour ne pas avoir besoin d'utiliser un corpus et donc simplifier le calcul. Seco et al. [Seco, N. et al. 2004] ont rapporté une valeur de corrélation de 0,84 entre les jugements humains de similarité et ceux donnés par leur mesure sur l'ensemble de données de Miller et Charles [Miller, G.A. and Charles, W.G. 1991]. Cette valeur est suggestivement proche de la limite supérieure de 0,88 postulée par Resnik dans [Resnik, P. 1999]. Cependant, la valeur $CI_{seco}$ est limitée car elle n'exploite qu'un seul type de relations. Dans le cas où on est intéressés par d'autres types de

relations, on peut utiliser une autre approche proposée par Pirró [Pirró, G. and Euzenat, J. 2010], qui étend l'approche de Seco et exploite d'autres types de liens.

#### 2.3.3.3 Similarité sémantique entre les groupes de concepts

Deux approches principales sont généralement distinguées pour calculer des mesures de similarité sémantique conçus pour la comparaison de deux ensembles de concepts [Harispe, S. et al. 2013] : (01) l'approche directe, et (02) l'approche indirecte.

***L'approche directe :***

L'approche directe correspond à une généralisation des approches définies pour la comparaison de paires de concepts afin de comparer directement deux ensembles de concepts. Il est à noter que deux ensembles de concepts peuvent être comparés par l'intermédiaire de leurs représentations vectorielles, par exemple, en utilisant la mesure de similarité cosinus. Néanmoins, celles-ci n'est pas significative dans la plupart des cas, puisque cette mesure ne tient pas compte de la similarité des éléments composant les ensembles comparés. un exemple d'une approche directe de calcul de similarité entre les groupes de concepts a été proposé dans [Pesquita, C. et al. 2007].

***L'approche indirecte :***

L'approche indirecte correspond aux mesures qui permettent d'évaluer la similarité entre deux ensembles de concepts en utilisant une mesure conçue pour la comparaison de paires de concepts. Ils sont généralement des simples agrégations de scores de similarités associés aux paires de concepts définies dans le produit cartésien des deux ensembles à comparés. Une stratégie indirecte en deux étapes peut être donc utilisée pour tirer profit des mesures existantes de comparaison des paires de concepts, dans le but de comparer des ensembles de concepts:

1. Calculer les similarités des paires de concepts obtenues à partir du produit cartésien des deux ensembles.
2. Les résultats de l'étape précédente sont ensuite agrégés en utilisant une stratégie d'agrégation, également appelée une stratégie de mixage (mixing strategy).

Des stratégies d'agrégation classiques peuvent être appliquées : *Max*, *Min*, *Moyenne*. Supposons qu'on a deux entités à comparer $X$ et $Y$, qui sont indexées par les concepts dans les deux ensembles $U$, $V$ respectivement. Une matrice de similarité $S$ est donc calculée. Cette matrice contient toutes les valeurs de similarité, paire à paire, des concepts dans $U$ et $V$. La matrice peut être calculée avec l'une des mesures de similarité mentionnées précédemment ($Sim_{Res}$, $Sim_{Lin}$, …etc). La matrice $S$ n'est pas nécessairement symétrique ou carré, car chaque entité peut être indexée avec plusieurs concepts différents. Les lignes et les colonnes de $S$

représentent deux comparaisons directionnelles différentes : les vecteurs de lignes correspondent à une comparaison de $U$ à $V$, et les vecteurs de colonnes de $V$ à $U$.

L'approche *Max* est donnée par la valeur maximale de similarité entre chaque concept dans l'ensemble $U$ et chaque concept dans l'ensemble $V$ :

$$Sim_{max}(U,V) = MAX_{u\in U, v\in V}\big(sim(u,v)\big) \tag{20}$$

La technique de la *Moyenne* ($Sim_{AVG}$) est donnée par la moyenne de similarité entre chaque concept dans $U$ et chaque concept dans $V$:

$$Sim_{AVG}(U,V) = \frac{\sum_{u\in U}\sum_{v\in V}\big(sim(u,v)\big)}{|U|+|V|} \tag{21}$$

D'autres stratégies plus raffinées ont également été proposées. Parmi les plus utilisées, on peut citer: *La moyenne maximale* $Sim_{AVGMAX}$ [Schlicker, A. et al. 2006], *le meilleur résultat avec similarité maximale* $Sim_{BMM}$ (pour *Best Match Max*) [Schlicker, A. et al. 2006], et *le meilleur résultat avec similarité moyenne* $Sim_{BMA}$ (pour *Best Match Average*) [Schlicker, A. et al. 2006, Pesquita, C. et al. 2008].

Soit l'ensemble des meilleurs résultats pour la comparaison de $U$ avec $V$, déterminé comme étant le vecteur des valeurs maximales dans les lignes de la matrice de similarités $S$. De même, les meilleurs résultats pour la comparaison de $V$ avec $U$ sont déterminés comme étant le vecteur des valeurs maximales dans les colonnes de la matrice de similarités $S$. Les moyennes de ces deux vecteurs de valeurs maximales représentent, respectivement, l'indice de similarité pour la comparaison de $U$ avec $V$ (i.e. $rowScore$) et l'indice de similarité pour la comparaison de $V$ avec $U$ (i.e. $columnScore$), tel que $rowScore \in [0,1]$ et $columnScore \in [0,1]$. Ainsi, pour les lignes on obtient :

$$\text{rowScore} = Sim_{AVGMAX}(U,V) = \frac{1}{|U|}\sum_{u\in U} max_{v\in V} sim(u,v) \tag{22}$$

Pour les colonnes on obtient :

$$\text{columnScore} = Sim_{AVGMAX}(V,U) = \frac{1}{|V|}\sum_{v\in V} max_{u\in U} sim(u,v) \tag{23}$$

La technique du meilleur résultat avec similarité moyenne, i.e. *BMA,* est une variante de $Sim_{AVGMAX}$ qui consiste à combiner les deux indices de similarité $rowScore$ et $columnScore$. Donc, la mesure $Sim_{BMA}$ est donnée par la similarité moyenne entre chaque concept $u \in U$ et son concept le plus similaire dans $V$, tout en calculant la moyenne avec son réciproque pour obtenir un score symétrique:

$$Sim_{BMA}(U,V) = \frac{\text{rowScore} + \text{columnScore}}{2} \tag{24}$$

La technique Best Match Max BMM [Schlicker, A. et al. 2006] est une autre variante de $Sim_{AVGMAX}$ qui vise à combiner les deux indice $rowScore$ et $columnScore$ en calculant leur maximum.

$$Sim_{BMM}(U,V) = \max(\text{rowScore}, \text{columScore}) \quad (25)$$

Selon Pesquita et al. [Pesquita, C. et al. 2008], il y a des limites dans l'utilisation de l'approche moyenne $Sim_{avg}$ et l'approche maximale $Sim_{max}$. En effet, La technique de la moyenne est imprécise dans le cas où $U$ et $V$ partagent plusieurs concepts. Par exemple: supposons qu'on a deux entités à comparer $X$ et $Y$ qui sont identiques (en termes de concepts) tel que $U = V = \{c_1, c_2\}$, et la similarité entre les deux concepts $c_1$ et $c_2$ est $sim(c_1, c_2) = 0$. Ainsi, en utilisant la technique de la moyenne on obtient une similarité $Sim_{avg}(U,V) = 50\%$ au lieu de d'avoir 100%, car les similarités sont calculées entre tous les paires de concepts possibles dans $U$ et $V$. En revanche, la technique du maximum est indifférente au nombre de concepts non liés dans $U$ et $V$. Par exemple: si $U = \{c_1, c_2\}$ et que $V = \{c_1\}$ on obtient alors une similarité $Sim_{max}(U,V) = 100\%$ malgré que $X$ et $Y$ ne sont clairement pas identiques à 100%. La technique *BMA*, quant à elle, ne souffre pas de ces limitations, et prend en considération les deux cas.

## 2.4 Concepts clés

Dans cette section nous présentons les concepts principaux en rapport avec la localisation de fonctionnalités, la compréhension des programmes, et la rétro-ingénierie.

### 2.4.1 Qu'est-ce que la Compréhension de Programmes ?

Il n'y a pas une définition commune pour la compréhension de programmes dans la littérature. Nous citons quelques définitions représentatives:

- « *La compréhension du programme est le processus d'acquisition de connaissances sur un programme d'ordinateur. La connaissance accrue permet d'effectuer d'autres activités telles que la correction des bogues, l'amélioration, la réutilisation et la documentation* » [Rugaber, S. 1995].
- *« Le processus de compréhension de programme utilise les connaissances existantes pour acquérir de nouvelles connaissances qui répondent finalement aux objectifs d'une tâche de cognition de code*» [Von Mayrhauser, A. and Vans, A.M. 1995].

En résumé, la compréhension peut être vue comme étant le processus qui extrait, à partir du programme, une connaissance conceptuelle à un haut niveau d'abstraction appelée les concepts abstraits du programme. Cette connaissance est utilisée dans plusieurs activités de redéveloppement de logiciel tel que la

documentation, la localisation de fonctionnalités, l'aide à la maintenance et la découverte des conceptions.

#### 2.4.1.1 La localisation de fonctionnalités

Dans les systèmes logiciels, une fonctionnalité représente une caractéristique qui est définie dans les exigences et qui est accessible aux développeurs aussi bien qu'aux utilisateurs. La maintenance et l'évolution des logiciels impliquent l'ajout de nouvelles fonctionnalités aux programmes, l'amélioration de fonctionnalités existantes, la suppression des bogues, ou bien la suppression des fonctionnalités non désirées. Les études ont montré que 20% du coût de développement est généré lors de la phase de développement, 40% lors de la phase de correction et 40% lors de la phase d'ajout de nouvelles fonctionnalités. C'est donc bien lors de la phase de correction-maintenance qu'il y a le plus d'opportunités d'obtenir des gains de productivité importants. Pour ce faire, il est essentiel de comprendre les programmes et en particulier leurs comportements. Cette opération peut coûter beaucoup en temps d'où la nécessité d'automatiser certains de ses aspects afin de réduire les coûts de production.

A ce propos, la construction d'une relation entre les concepts qui sont dans l'esprit du programmeur et les composants du programme est l'une des approches qui facilite la compréhension et la maintenance des programmes. L'identification d'un emplacement initial dans le code source qui correspond à une fonctionnalité spécifique est connu comme la localisation de fonctionnalités (ou de concepts) [Biggerstaff, T.J. et al. 1993, Rajlich, V. and Wilde, N. 2002]. Ceci peut être réalisé en établissant la correspondance entre les concepts (de haut niveau) et les éléments de code source au moyen de vocabulaires utilisés par ces derniers.

#### 2.4.1.2 La documentation

La documentation a pour but de créer une représentation sémantiquement équivalente et au même degré d'abstraction que la représentation originale [Chikofsky, E.J. and Cross, J.H. 1990]. Comme exemple de documentation, il est possible de prendre un diagramme de classes créé à partir d'un outil pour représenter le code source d'un logiciel. Il sera sémantiquement équivalent et représentera le même niveau d'abstraction, car ce diagramme ne fait que représenter le code source sous une autre forme.

#### 2.4.1.3 La redécouverte de la conception

L'objectif de la redécouverte de la conception est de créer des représentations à un plus haut niveau d'abstraction [Chikofsky, E.J. and Cross, J.H. 1990]. Selon Biggerstaff, « *elle recrée une abstraction de la conception à l'aide du code source, de la documentation de conception existante (si disponible), de l'expérience personnelle, des connaissances générales au sujet du problème et du domaine du*

*logiciel* » [Biggerstaff, T.J. 1989]. Ainsi, en plus de décrire comment le logiciel est conçu, ce type de compréhension de logiciel utilise des techniques permettant de créer des abstractions afin de faciliter la compréhension.

#### 2.4.1.4 L'aide à la maintenance

Le processus de maintenance consiste à modifier le logiciel après sa livraison afin de corriger des problèmes (maintenance corrective), améliorer certaines de ses caractéristiques (maintenance perfective), l'adapter à des changements de son environnement (maintenance évolutive) ou même le réorganiser pour faciliter les changements éventuels (maintenance préventive) [Chikofsky, E.J. and Cross, J.H. 1990]. Il est évident alors que la compréhension de ce que le logiciel réalise et la façon dont il le fait apporte un aide direct pour la tâche de maintenance. En revanche, l'analyste du logiciel a besoin d'un procédé et d'outils nécessaire pour arriver à comprendre le logiciel. A ce sujet, la rétro-ingénierie est une pratique qui fournit des supports et des techniques pour faciliter la compréhension des programmes et, par conséquent, leur maintenance.

### 2.4.2 La rétro-ingénierie

La rétro-ingénierie d'un logiciel est « *le processus d'analyse d'un logiciel ayant pour objectif d'identifier les composantes et les relations entre les composantes de ce logiciel et de lui créer une représentation dans une autre forme ou à un niveau d'abstraction plus élevé* » [Chikofsky, E.J. and Cross, J.H. 1990]. Mais si le système est moindrement gros ou complexe, la représentation ne serait pas tellement plus facile à comprendre que le code source. C'est ici que le concept d'abstraction prend son importance. Faire une abstraction consiste à retirer les détails inutiles à une tâche de compréhension particulière.

La pratique de la rétro-ingénierie est donc très précieuse pour la compréhension du logiciel comme l'affirme Chikofsky: « *L'essentiel objectif de la rétro-ingénierie d'un système logiciel est d'augmenter la compréhensibilité globale du système pour l'entretien aussi bien que le nouveau développement*» [Chikofsky, E.J. and Cross, J.H. 1990]. Chikofsky [Chikofsky, E.J. and Cross, J.H. 1990] a aussi distingué entre deux concepts : (01) la redocumentation, et (02) la redécouverte de la conception, que nous avons présenté dans section précédente, comme étant deux sous-domaines de la rétro-ingénierie.

Le terme *rétro-ingénierie* est parfois confondu avec le terme *réingénierie*. La réingénierie est « l'examen et la modification d'un système afin de le reconstituer et le réaliser sous une nouvelle forme » [Chikofsky, E.J. and Cross, J.H. 1990]. Le but de la réingénierie est donc d'analyser et comprendre un système (ou une partie du système) afin de le reconcevoir sous une nouvelle forme. La réingénierie implique donc un processus de rétro-ingénierie (i.e. détection et analyse) suivi d'un

processus d'ingénierie (i.e. transformation). Autrement dit, la rétro-ingénierie fait partie du processus de réingénierie. Dans le contexte des lignes de produits logiciels, le processus de réingénierie consiste à l'extraction d'une ligne de produits logiciels à partir des systèmes existants. Le but de notre travail de thèse consiste donc à la rétro-ingénierie de l'ensemble de variabilités et de commonalités d'une LdP à partir du code source d'un ensemble prédéfini de systèmes.

#### 2.4.2.1 Les données en entrées pour la rétro-ingénierie

La rétro-ingénierie peut être effectuée à partir de n'importe quel niveau d'abstraction ou à n'importe quelle étape du cycle de vie [Chikofsky, E.J. and Cross, J.H. 1990]. La rétro-ingénierie peut alors se baser sur plusieurs sources d'information afin de réaliser ses objectifs. Ces sources de données peuvent prendre plusieurs formes : le code source, les bases de connaissances, les traces d'exécution, l'exécutable, la documentation et même les intervenants dans la vie d'un logiciel tels que les acteurs et les experts du domaine. Cependant, le code source est considéré généralement comme la représentation la plus fiable du logiciel contrairement à d'autres, comme les documents de conceptions qui n'ont peut-être pas été respectés lors de l'implémentation. C'est pourquoi il est la source d'information la plus fréquemment employée. À ce propos, il existe plusieurs techniques pour analyser le code source d'un programme allant des approches statiques complexes jusqu'aux approches d'analyse dynamique de l'exécution de programmes.

#### 2.4.2.2 Analyse statique

L'analyse statique implique en général l'analyse des informations obtenues directement en examinant le code source du logiciel visé, pour dériver les propriétés qui se tiennent pour toutes les exécutions [Ball, T. 1999, Ernst, M.D. 2003]. Cette analyse permet d'examiner les liens entre les composantes du code source. En général, l'analyse statique est *conservatrice* et *solide* :

- ***La solidité*** (anglicisme : *soundness*) garantit que les résultats de l'analyse sont une description précise du comportement du programme, indépendamment des données d'entrée et de l'environnement dans lequel s'exécute le programme [Ball, T. 1999, Ernst, M.D. 2003].
- ***Le conservatisme*** signifie qu'il est possible de rapporter des propriétés faibles, en préservant ainsi la fiabilité, cependant ces propriétés peuvent ne pas être assez fortes pour être utiles [Ball, T. 1999, Ernst, M.D. 2003].

Puisqu'il y a beaucoup d'exécutions possibles, il n'est pas raisonnable de considérer chaque état possible du programme. Par conséquent, l'analyse statique utilise généralement un modèle abstrait de l'état du programme qui perd une partie de

l'information, mais qui est plus compact et plus facile à manipuler [Ernst, M.D. 2003].

#### 2.4.2.3 Analyse dynamique

Contrairement à l'analyse statique, l'analyse dynamique dérive des propriétés qui se tiennent pour une ou plusieurs exécutions en examinant le programme en cours d'exécution [Ball, T. 1999, Ernst, M.D. 2003]. L'analyse dynamique est fréquemment utilisée quand on essaie de comprendre les performances et l'exactitude des propriétés du programme, ou lorsqu'on a besoin de comprendre les caractéristiques d'une conception.

Bien que l'inconvénient de l'analyse dynamique réside dans le fait qu'on ne puisse pas généraliser les résultats obtenus pour toutes les exécutions futures [Ernst, M.D. 2003], son utilité vient de deux de ses caractéristiques essentielles [Ball, T. 1999]:

- **La précision de l'information** : l'analyse dynamique implique généralement l'instrumentation, qui consiste à ajouter des instructions au programme, pour examiner ou enregistrer certains aspects pendant son exécution. Cette instrumentation peut être ajustée pour collecter précisément les informations nécessaires pour régler un problème particulier. Par exemple, pour analyser la forme des structures de données créées par un programme (listes, arbres, graphe, …etc.), un outil d'instrumentation peut être créé pour enregistrer les liens entre les cellules de stockage dans l'espace mémoire alloué.
- **La dépendance vis à vis les entrées du programme**: bien que cette dépendance rend l'analyse dynamique incomplète, elle fournit également un mécanisme puissant pour relier les entrées et les sorties d'un programme avec son comportement. Avec l'analyse dynamique, il est simple de relier les modifications sur les données d'entrée d'un programme aux changements de son comportement interne et des résultats fournis en sortie, puisqu'ils sont tous directement observables et liés par l'exécution du programme.

#### 2.4.2.4 Relation entre l'analyse dynamique et l'analyse statique

L'analyse dynamique et l'analyse statique sont deux techniques complémentaires dans certains points:

- **La synergie :** En général, les analyses dynamiques génèrent « *des invariants* » du programme tandis que l'analyse statique peut aider à déterminer si ces invariants sont vrais pour toutes les exécutions de programme [Ball, T. 1999, Ernst, M.D. 2003, Mock, M. 2003]. Un invariant est une propriété d'un programme qui peut être maintenue (vraie) à un ou

plusieurs points du programmes [Ernst, M.D. 2000]. Un point est un emplacement arbitraire dans un programme. Deux exemples de points sont les *entrées* et les *sorties* de méthodes. Ainsi, les invariants détectés à l'entrée et à la sortie d'une méthode correspondent aux pré-conditions et post-conditions de cette dernière [Perkins, J.H. and Ernst, M.D. 2004].

- **La portée :** l'analyse dynamique a le potentiel de découvrir les dépendances sémantiques, largement séparées dans le temps, entre les entités du programme [Ball, T. 1999]. Par exemple, pour comprendre les systèmes concurrents on doit comprendre les interactions entre les différents processus, par exemple, comment un évènement dans le contexte d'un processus pour affecter les évènements dans le contexte des autres processus. ces interactions se produisent seulement au moment de l'exécution et ne peuvent pas être examinées statiquement dans le code [Stroulia, E. and Systä, T. 2002].
- **La précision :** L'analyse dynamique a l'avantage d'examiner le domaine concret de l'exécution du programme sans aucune approximation ou abstraction [Ernst, M.D. 2003]. L'analyse statique doit assurer cette abstraction afin de garantir la terminaison de l'analyse, en limitant ainsi l'information dès le début. L'abstraction peut être une technique utile pour réduire les temps d'exécution de l'analyse dynamique et de réduire la quantité d'informations enregistrées, dont on a pas besoin [Ball, T. 1999]. En effet, elle est pratique dans le cas où l'analyste est intéressé seulement par une partie spécifique du logiciel ; ça ne sert à rien alors de générer un nombre considérable de traces d'exécution pour le système entier [Stroulia, E. and Systä, T. 2002].

Tandis que l'analyse dynamique devrait généralement être complétée par une analyse statique pour être applicable, les outils de génie logiciel peuvent profiter des avantages de l'analyse dynamique dans de nombreux cas, même sans utiliser l'analyse statique. En effet, dans de nombreux cas les résultats imprécis de l'analyse dynamique peuvent être plus utiles que les résultats fiables d'une analyse statique, dans le cas où cette dernière rapporte à l'analyste beaucoup de résultats qui ne sont pas utiles dans la pratique [Mock, M. 2003].

D'autre part, en raison des changements dans la maniéré avec laquelle les logiciels sont écrits et déployés, l'efficacité de l'analyse statique est décroissante [Mock, M. 2003]. En effet, de nos jours, les nouveaux logiciels sont souvent orientés objet, orienté aspect, orientés service ou bien orientés composant, et les aspects dynamiques comme, par exemple, le polymorphisme ne peuvent être détectés facilement avec une analyse statique. Ils imposent de procéder à une analyse dynamique.

Bref, l'importance de l'analyse dynamique continue à augmenter, et le besoin à ses méthodes ne cessent d'accroitre, fournissant ainsi un bon moyen pour améliorer les outils qui aident les programmeurs dans la compréhension, la maintenance et l'évolution des logiciels.

## 2.5 Conclusion

Dans ce chapitre, nous avons présenté le contexte de notre travail ; nous avons abordé tous les concepts de base de l'ingénierie des LdP. Nous avons aussi présenté les concepts clés des techniques utilisées dans notre approche : clustering, similarité sémantique, mesures d'évaluations, …etc.

# Chapitre 3: ETAT DE L'ART

Dans ce chapitre, nous présentons l'état de l'art en relation avec nos contributions. La section 3.1 présente l'ensemble des études effectuées dans le domaine de localisation de fonctionnalités en général. La section 3.2 illustre les approches spécifiques à la réingénierie des LdP. Enfin, la section 3.3 conclut ce chapitre.

## 3.1 La localisation des fonctionnalités

Ratiu et Deissenboeck [Ratiu, D. and Deissenboeck, F. 2006, Ratiu, D. and Deissenboeck, F. 2007] ont proposé une approche semi-automatique qui permet de mettre en correspondance les concepts du monde réel avec les parties pertinentes du code source. Leur approche vise l'interprétation des programmes du point de vue des connaissances du domaine qu'ils implémentent, qui peuvent être des fonctionnalités. Ils ont développé un framework qui décrit les défauts sémantiques causés par la nomination impropre (ex. synonymes, polysémies, …etc), ainsi qu'un algorithme pour récupérer les correspondances entre les éléments d'une ontologie et les éléments du programme en utilisant l'appariement de graphes (graph matching) ; les concepts sont représentés dans l'ontologie et les programmes sont modélisés sous forme de graphes. Les correspondances extraites automatiquement sont validées manuellement par un utilisateur. De plus, l'ontologie utilisée est construite manuellement car, selon les auteurs, les ontologies actuellement disponible (sur-étagères) ne couvrent que des parties restreintes de certains domaines. En outre, l'approche vise à étudier seulement la partie du domaine reflétée par le programme à analyser. Le framework et l'algorithme ont été appliqués à un API standard Java, et ont révélé des exemples concrets de défauts sémantiques. Ratiu et al. [Ratiu, D. et al. 2008] ont étendu cette idée pour prendre en charge plusieurs API à la fois, au lieu d'analyser un seul programme. En effet, leur approche (semi-automatique) consiste à analyser plusieurs API, qui couvrent le même domaine, afin d'en extraire une abstraction du domaine, i.e. une ontologie, qui généralise sur les connaissances contenues dans ces API. Ils ont étudié d'abord la portée de quelques API en calculant la similarité entre les identificateurs utilisés dans le code. Ensuite, chacun des API doit être abstrait sous forme d'un graphe. Finalement, après avoir analysé ces graphes en utilisant la technique du *graph matching*, l'ontologie du domaine est alors construite à partir des concepts communs entre les graphes selon leurs pondérations calculées durant le matching (nombre de participation dans un matching). Les ontologies obtenues avec cette technique peuvent être donc utilisées comme entrée pour d'autres activités d'analyse de programmes et de réingénierie.

Abebe et Tonella [Abebe, S.L. and Tonella, P. 2010] ont introduit une approche qui vise à extraire des concepts à partir du code source en utilisant le TALN (Traitement Automatique du Langage Naturel). Les identificateurs d'éléments du programme

sont extraits et utilisés pour former des phrases candidates, tout en éliminant les phrases qui ne suivent pas certaines règles. Les phrases restantes sont utilisées comme entrée pour créer des ontologies qui capturent les concepts et les relations à partir du code source ; Les identificateurs représentent les concepts et la sémantique des phrases établit les relations entre ces concepts. L'ontologie automatiquement extraite est utilisée, par la suite, pour formuler des requêtes plus précises afin de réduire l'espace de recherche durant le processus de localisation de fonctionnalités. Les expérimentations menées en utilisant cette approche montrent une augmentation dans la précision des requêtes dans plus de 50% des cas examinés ; en moyenne, l'augmentation de précision peut être quantifiée comme d'environ moins 86% de fichiers à inspecter. Un autre avantage de cette approche et l'expansion automatique des abréviations durant l'extraction des termes à partir des identificateurs d'éléments du programme. Abebe et Tonella ont proposé, dans une autre étude [Abebe, S.L. and Tonella, P. 2011], quelques améliorations à leur approche initiale. En effet, l'ontologie automatiquement générée semble être d'une taille très importante et contient, en plus des concepts du domaine, des concepts techniques qui sont spécifiques à l'implémentation (bruits). Ainsi, les auteurs ont proposé une approche semi-automatique afin de créer une liste de mots-clés, en utilisant toujours le code source comme un jeu d'apprentissage. Dans une deuxième étape, l'utilisateur doit sélectionner à partir de cette liste les concepts du domaine qui lui semblent adéquats. Ainsi, lors de l'étape de filtrage, on ne garde que les concepts constitués de mots-clés sélectionnés par l'utilisateur ainsi que leurs interrelations, réduisant ainsi le nombre de milliers à quelques dizaines de concepts seulement. L'expérimentation menée par Abebe et Tonella utilisant ces filtres montrent des résultats prometteurs avec une bonne précision. De plus, leur technique de filtrage est indépendante de l'étape de génération d'ontologies et peut, donc, compléter n'importe quelle autre approche.

L'approche de Abebe et Tonella est similaire à celle de Petrenko et al. [Petrenko, M. et al. 2008], mais la principale différence est que la première approche génère les ontologies automatiquement, alors que pour cette dernière approche les ontologies sont générées manuellement par les développeurs. En outre, les fragments d'ontologies initialement construites dans l'approche de Petrenko et al. doivent faire l'objet de plusieurs raffinements (ex. renommage, addition, ou suppression des concepts) en inspectant la documentation, l'interface de l'application analysée, ou même en observant son comportement pour découvrir des détails additionnels. Ainsi, les ontologies construites manuellement peuvent (ou pas) être d'une grande précision selon les compétences des développeurs.

## 3.2 La rétro-ingénierie des LdP

Il existe plusieurs études dans la littérature qui ont mis l'accent sur l'approche extractive des LdP. Chacune de ces études a abordé un problème spécifique, allant

de la rétro-ingénierie des modèles de LdP et leurs contraintes, à la localisation de fonctionnalités dans le code source et/ou dans les documents de spécifications, jusqu'à la restructuration des LdP existantes (code et/ou modèles). Ces approches peuvent être classées selon leurs finalités en deux axes principaux : (01) la réingénierie d'une nouvelle LdP à partir d'anciens systèmes, et (02) la restructuration des LdP existantes [Lozano, A. 2011, Laguna, M.A. and Crespo, Y. 2013]. Une autre classification des approches existantes peut être donnée aussi en termes d'entrées utilisées lors du processus du rétro-ingénierie, et on distingue alors : (a) les techniques basées sur la documentation, et (b) les techniques basées sur le code source. Nous avons choisi d'illustrer les techniques qui semblent être proches de notre travail et nous nous limitons donc au deuxième type de classification, i.e. en termes de ressources utilisées.

### 3.2.1 Les approches basées sur la documentation

Le travail de Graaf et al. [Graaf, B. et al. 2006] consiste à faire migrer une architecture existante vers une LdP via une transformation automatique de modèles. Les règles de transformation utilisées sont définies en utilisant ATL (*Atlas Transformation Language*). Ainsi, la migration ne peut être possible que si la variabilité est définie par un méta-modèle.

L'étude effectuée par Niu [Niu, N. and Easterbrook, S. 2008] introduit une approche basée sur la segmentation, la théorie de l'information, et le TALN. Cette approche permet d'extraire les caractéristiques fonctionnelles depuis les documents textuels d'exigences, selon le point de vue utilisateur (vue externe) ainsi que le point de vue du concepteur (vue interne). Les auteurs ont utilisé un algorithme de segmentation **OPC** (***O****verlapping* ***P****artitionning* ***C****luster*) qui facilite l'identification du chevauchement des aspects (*crosscutting concerns*) et minimise la perte de l'information. La théorie de l'information a été utilisée pour reconstruire, à partir des clusters, une hiérarchie et effectuer, par la suite, une fusion bottom-up en calculant à chaque fois la perte de l'information engendrée par cette fusion. Un MF est ensuite construit, et les fonctionnalités sont jugées comme obligatoires ou optionnelles selon le ratio de leurs occurrences par rapport à l'occurrence de leurs parents (calculé durant la fusion). Une technique de TALN a été utilisée pour définir la similarité entre les attributs des FRP (*functional requirement profiles*) qui sont créés manuellement. L'avantage de cette approche est qu'elle peut fournir plusieurs vues, avec différentes granularités, selon les paramètres de l'algorithme de segmentation fixés par l'utilisateur. Néanmoins, l'approche ne permet pas la détection des contraintes entre les fonctionnalités.

Rashid et al [Rashid, A. et al. 2011] ont proposé une autre technique basée sur le traitement automatique du langage naturel (TALN) et la segmentation. Cette technique peut aider les ingénieurs en automatisant la construction de modèles de

fonctionnalités à partir des documents d'exigences. Les techniques de TALN utilisées permettent non seulement l'extraction de fonctionnalités mais aussi l'identification des similarités ainsi que les variabilités contenues dans des documents hétérogènes. Une première expérience menée en utilisant cette technique a montré que 50% de fonctionnalités du modèle généré avaient une très grande précision. Néanmoins, le modèle généré était d'une grande taille par rapport à celui qui a été créé manuellement. D'où, l'intervention d'un expert est toujours nécessaire pour effectuer des pré/post-traitements. Les auteurs ont réclamé que ce problème est causé par l'algorithme de segmentation utilisé ainsi que l'information inutile contenue dans les documents d'exigences.

Haslinger et al. [Haslinger, E.N. et al. 2011, Haslinger, E.N. et al. 2013] ont présenté un algorithme qui reconstruit un MF de base à partir des ensembles de fonctionnalités qui décrivent les caractéristiques fournis par leurs systèmes. L'idée est de trier la totalité de fonctionnalités dans un tableau et de construire, récursivement, le MF d'une manière descendante en recalculant à chaque itération le tableau de fonctionnalités. Les fonctionnalités utilisées ont été obtenues à partir d'un dépôt en ligne en décomposant les modèles récupérés en fonctionnalités prêtes à être utilisées directement sans aucun prétraitement. Les expérimentations ont montré que les MF de base calculés par cet algorithme ont été identiques aux modèles initiales récupérés depuis le dépôt.

Ryssel et al. [Ryssel, U. et al. 2011] ont proposé une approche pour extraire des MF à l'aide de l'analyse de concepts formels (ACF), en utilisant comme entrée une matrice d'incidence qui contient des relations de correspondance, cette matrice peut être obtenus utilisant par exemple l'approche dans [Ryssel, U. et al. 2012]. La matrice décrit les parties d'un ensemble de modèles *orientés bloc de fonction* (*function-block-oriented models*[3]). En effet, ce type de modèles est similaire aux modèles de composants sauf que ce dernier est orienté services tandis que les modèles utilisés par Ryssel et al. sont orientés flux de données. Cette approche est capable de dériver des modèles de fonctionnalités complets qui peuvent indiquer si une fonctionnalité est optionnelle, alternative (XOR-groupe), ou de type OR (OR-group). La principale différence entre cette approche et d'autres approches basées sur ACF est l'utilisation de méthodes optimisées de telle façon que les lattices, les implications, et les modèles de fonctionnalités sont générés en très peu de temps.

### 3.2.2 Les approches basées sur le code source

Ziadi et al. [Ziadi, T. et al. 2012] ont proposé une approche automatique pour l'identification de fonctionnalités à partir du code source d'un ensemble de variantes logicielles. Cette approche suppose que les variantes de produits utilisent le même vocabulaire pour nommer les packages, les classes, les variables de classes

[3] Un type de modèle d'architectures pour les systèmes embarqués

et les méthodes dans le code source. Cette approche fait d'abord l'abstraction des produits analysés en construisant un modèle UML pour chaque variante logicielle, i.e. un diagramme de classe simplifié. Ensuite, des ensembles de primitives de construction (des éléments atomiques), dénotés *CP*, sont générés à partir des diagrammes de classes UML. Un algorithme a été proposé pour identifier les fonctionnalités candidates à partir de ces ensembles de *CP*; l'algorithme identifie les éléments atomiques qui semblent identiques dans les produits disponibles. En dernier lieu, des post-traitements sont effectués manuellement sur l'ensemble de candidats, i.e. la suppression des candidats non pertinents et l'ajout de fonctionnalités manquantes, pour produire l'ensemble final de fonctionnalités de la LdP. Ce dernier va servir de base pour construire un modèle de fonctionnalités. Les premières expérimentations utilisant cette approche sur des variantes de l'outil ArgoUML[4] ont données des résultats prometteurs. Cependant, étant donné que cette approche utilise des *CP* qui sont construits à partir d'un diagramme de classes UML (abstraction), les éléments de code qui sont définis dans le corps d'une méthode donnée ne sont pas pris en charge. De plus, cette approche suppose que l'ensemble maximal des *CP* (qui sont partagés par tous les variantes de produits) est considéré comme une seule fonctionnalité obligatoire (i.e. une commonalité).

Les auteurs dans [Paskevicius, P. et al. 2012] ont proposé une approche qui permet de dériver un modèle de fonctionnalités à partir du bytecode Java en utilisant l'analyse statique et la segmentation. Pour ce faire, ils ont d'abord supposé que chaque méthode (resp. variable) représente une *fonctionnalité atomique*, et que chaque classe (resp. package) représente une fonctionnalité *composite*. Ensuite, ils ont essayé d'extraire un graphe de dépendances entre ces fonctionnalités et l'utiliser pour construire une matrice de dissimilarités. La matrice résultante doit être analysée, ensuite, avec l'algorithme *CobWeb* [Fisher, D. 1987] implémenté dans l'API WEKA[5] pour créer une hiérarchie de fonctionnalités. Cette hiérarchie est utilisée pour créer une description du MF final en Prolog ainsi qu'en FDL (Feature Description Language) [Van Deursen, A. and Klint, P. 2002]. Cette approche peut être utile durant la configuration partielle d'un système; lorsqu'on veut dériver une version allégée du système. Néanmoins, cette approche ne prend pas en charge toutes les notations des MF telles que les contraintes de type *inclure* et *exclure*, et les fonctionnalités dérivées peuvent être marquées seulement comme *Alternative* (seulement la racine est étiquetée comme *obligatoire*). Ce choix conceptuel des auteurs est justifié par le fait que l'approche permet d'analyser le code d'un seul système qui représente une seule combinaison possible des fonctionnalités.

Al-Msie'deen et al. [Al-Msie'Deen, R. et al. 2013] ont proposé une approche qui permet de dériver un MF à partir du code source orienté objet. Leur approche

[4] http://argouml.tigris.org/
[5] http://www.cs.waikato.ac.nz/ml/weka/

consiste à explorer d'abord les programmes candidats pour en extraire les éléments de code source (OBEs) tels que des packages, des classes, méthodes, attributs…etc. Ces éléments sont ensuite analysés utilisant ACF pour identifier un bloc d'éléments communs (CB) qui contient les éléments partagés par toutes les variantes, ainsi que des blocs de variations (BVs) qui contiennent les éléments partagés par certaines variantes. Par la suite, le CB (resp. BVs) est analysé en combinant l'analyse ACF et la ercherche d'information (i.e. l'indexation sémantique latente LSI) et la similarité structurelle pour extraire des blocs atomiques communs CABs (resp. ABVs) qui représentes les commenalités (resp. variabilités) potentielles. L'utilisation de LSI est basée sur l'hypothèse selon lequel les OBEs impliqués dans l'implémentation d'une caractéristique fonctionnelle sont lexicalement plus rapprochés les uns des autres que dans le reste des OBEs. La similarité structurelle quant à elle consiste à l'utilisation de l'information dans le code source concernant les dépendances structurelles entre les classes pour augmenter la précision des résultats. Ainsi, le contexte formel utilisé dans l'analyse ACF est la combinaison des deux matrices de similarités : structurelle et lexicale. L'intérêt d'utiliser l'analyse ACF à cette étape est pour aider à extraire des concepts formels représentant des OBEs qui sont mutuellement similaires. Cette approche a donné de bons résultats. Néanmoins, les auteurs ont étudié les variantes de produits dans lesquelles la variabilité est représentée dans les packages ou les classes sans tenir compte de la variabilité représentée au niveau des méthodes ou au niveau du corps des méthodes. De plus, les développeurs peuvent ne pas utiliser les mêmes vocabulaires pour nommer les OBEs à travers toutes les variantes logicielles. Cela signifie donc que la similarité lexicale ne peut pas être toujours fiable.

Le tableau *Tab. 3.1* donne un bref résume des différentes caractéristique d'approches extractives que nous avons illustrées dans cette section.

### 3.2.3 Synthèse

Comme nous pouvons le remarquer dans la section précédente, la plupart des approches existantes qui abordent le problème de rétro-ingénierie des MF sont semi-automatiques et utilisent la documentation et les descriptions textuelles comme données d'entrées. Cependant, une telle pratique comporte plusieurs défis. Bien que le type de donnée en entrée dans une approche soit fortement lié à son objectif et son type d'usage, le niveau d'abstraction et de formalisation de ces données doivent être aussi pris en considération. Zaidi et al. [Ziadi, T. et al. 2012] utilisent en entrée des parties de diagrammes de classes UML qui sont générés à partir du code source. Cependant, une telle abstraction est susceptible d'omettre beaucoup de détails liés à la variabilité qui peuvent être exprimés, par exemple, à l'intérieur d'une méthode. Les MF construits à partir des spécifications textuelles dans [Rashid, A. et al. 2011] étaient d'une taille importante et souffrent des imprécisions. Les auteurs justifient leurs conclusions par la nature hétérogène et

surtout textuelle des entrées. Ces derniers sont susceptibles d'être remplis de langages imprécis habituellement utilisés dans les conversations. Outre que l'ambiguïté, la scalabilité représente un problème dans le contexte de LdP. En effet, nous pouvons trouver un nombre important de documents associés à une variante logicielle, dont chacun est d'une taille importante.

| | **Al-Msie'deen et al. 2013** | **Ziadi et al. 2012** | **Paskevicius et al. 2012** |
|---|---|---|---|
| **Phase de réingénierie** | Détection + analyse | Détection | Détection + analyse |
| **Technique utilisée** | Analyse statique, recherche d'information (ACF, LSI) | Recherche d'information (fréquence des termes) | Analyse statique, clustering |
| **Nombre de variantes logicielles** | Plusieurs variantes logicielles | Plusieurs variantes logicielles | Un seul produit logiciel |
| **Hétérogénéité des variantes logicielles** | Plusieurs produits utilisant le même vocabulaire | Plusieurs produits utilisant le même vocabulaire | Un seul produit logiciel homogène |
| **Granularité de la variabilité** | Classes, packages | Classes, méthodes, variable de classe. | Classes, méthodes, variables de classe, corps des méthodes |
| **Type de similarité** | Structurelle | Structurelle | Structurelle |
| **Données d'entrée** | Code source Java | Code source | Byte code Java |
| **Données de sorties** | Commonalités + Variabilités | Une seule commonalité + variabilités | Commonalités + variabilités |
| **Chevauchement de fonctionnalités** | Oui | Non | Non |
| **Contraintes complexes : inclure, exclure** | Non | Non | Non |
| **Contraintes de sélection (OR, XOR, obligatoire)** | OR, obligatoire | Non | XOR, obligatoire |

**Tab 3.1 : Une comparaison entre les approches d'extraction des LdP à partir du code source.**

D'autres travaux, tel que celui de Haslinger et al. [Haslinger, E.N. et al. 2011], utilisent un ensemble de MF des systèmes existants afin d'en extraire le MF global de la LdP. Cependant, les systèmes existants ne disposent pas toujours de MF, et

même s'ils existent ils peuvent ne pas être à jour et, par conséquent, ne reflètent pas vraiment les variabilités de ces systèmes.

En ce qui concerne la technique utilisée, la plupart des approches utilisent l'analyse statique et/ou la recherche d'information. L'utilisation de la recherche d'information peut être justifiée par sa capacité de traiter une quantité massive de données afin d'en extraire de l'information pertinente. De plus, utilisant la recherche d'information, on peut traiter des documents/artéfacts logiciels d'un niveau d'abstraction élevé comme, par exemple, les exigences rédigées en langage naturel. Parmi les techniques de recherche d'information, la classification a attiré le plus d'attention, étant donné qu'elle est la plus appropriée au problème d'extraction des MF. Paskevicius et al. [Paskevicius, P. et al. 2012] considèrent la hiérarchie générée par l'algorithme Cobweb comme étant un MF, alors qu'il n'y a tout simplement pas assez d'informations dans les données en entrée afin de déterminer une seule hiérarchie préférée. De plus, l'utilisation d'un simple algorithme de partitionnement pour générer des groupes disjoints d'éléments de programme peut entraîner une perte d'information, car un élément de programme peut faire partie de plus d'une fonctionnalité (i.e. problème des préoccupations transversales). Niu et al. [Niu, N. and Easterbrook, S. 2008], ont abordé le problème de chevauchement en utilisant un algorithme de partitionnement avec chevauchement appelé OPC [Chen, Y.-L. and Hu, H.-L. 2006]. Néanmoins, l'algorithme OPC requiert quatre paramètres qui doivent être spécifiés par l'utilisateur, ce qui réduit considérablement l'automatisation de la tâche. Outre que le partitionnement, plusieurs chercheurs ont utilisé l'analyse ACF pour extraire des MF tout en prenant en compte le problème de chevauchement. Cependant, il existe une limite à l'utilisation de ACF. En effet, non seulement ACF ne garantit pas que les fonctionnalités générées (concepts formels) soient disjointes et couvrent l'ensemble des entités [Tonella, P. and Potrich, A. 2007], mais elle est également exposée au problème de la perte d'informations. Par exemple, dans[Al-Msie'Deen, R. et al. 2013], des matrices de similarités ont été transformées en un contexte formel (i.e. un contexte binaire) en utilisant un seuil fixe. La perte d'information causée par un tel usage de seuil fort peut affecter la qualité du résultat, comme il a été déjà expliqué par les auteurs dans [Al-Msie'deen, R. et al. 2014]. L'approche REVPLINE proposée par Al-Msie'Deen et al. [Al-Msie'Deen, R. et al. 2013, Al-Msie'deen, R. et al. 2014] génère des fonctionnalités de LdP en utilisant en entrées le code source des variantes logicielles. L'approche REVPLINE suppose que les produits analysés sont développés avec la technique du copier-coller, i.e. ils utilisent le même vocabulaire. Toutefois, si cette hypothèse ne tient pas, il est donc essentiel d'avoir un MF distinct pour chacun des produits analysés afin d'en extraire un modèle de LdP.

## 3.3 Conclusion

Dans ce chapitre nous avons dressé un état de l'art en explorant les différentes études en relation avec notre approche. Nous avons inspecté d'abord les différentes contributions dans le domaine de localisation de fonctionnalité. Ensuite, nous avons rapporté en détails les différentes études effectuées dans le domaine de rétro-ingénierie des LdP, tout en analysant ces études.

# Chapitre 4: EXTRACTION DE FONCTIONNALITES

Ce chapitre présente la première contribution de notre approche qui consiste à l'extraction de fonctionnalités d'un système logiciel à partir de son code source orienté objet. La section 4.1 présente l'idée globale derrière cette contribution. La section 4.2 explique, d'une manière détaillée, les différentes étapes du processus d'extraction. La section 4.3 explique les points forts et les points faibles de notre approche. Enfin, la section 4.4 conclut ce chapitre. Le travail présenté dans ce chapitre a fait l'objet d'une publication [Araar, I.E. and Seridi, H. 2016].

## 4.1 Objectif et hypothèses

L'objectif de l'approche proposée dans ce chapitre est d'identifier toutes les implémentations de fonctionnalités d'un produit logiciel donné, en se basant sur l'analyse statique du code source. En fait, nous reconnaissons qu'il est essentiel d'avoir, pour chaque logiciel, un catalogue de fonctionnalité qui est à jour et qui reflète les modifications qui ont été apportées au code source au fil du temps. Les fonctionnalités générées utilisant notre approche peuvent être utilisées pour documenter un système donné, ainsi que pour la rétro-ingénierie d'un MF d'une LdP. Nous adhérons à la classification donnée par Kang et al. [Kang, K. et al. 1990], qui distingue trois types de caractéristiques de systèmes logiciels: fonctionnelles, opérationnelles et caractéristiques de présentation. Dans la cadre du travail actuel, nous nous concentrons sur l'identification des caractéristiques fonctionnelles qui expriment la façon dont les utilisateurs peuvent interagir avec un système logiciel.

Les caractéristiques[6] fonctionnelles sont implémentées utilisant des éléments de programmes orientés objets (EdP), tels que les packages, les classes, les variables de classes, les méthodes et les éléments de corps des méthodes. Nous considérons également que les EdP peuvent être classés en deux catégories: (1) des éléments atomiques de programme (EAP), et (2) des éléments composites de programme (ECP). L'EAP est un élément de construction de base dans un programme, i.e. une variable ou une méthode. Un ECP est une composition d'EdP atomiques et/ou composites (i.e. une classe ou un package). Une dépendance est une relation entre deux EdP. Un élément A dépend de B si A fait référence à l'élément B. Par exemple une méthode A() utilise une variable B ou fait appelle à une méthode B(). Etant donnée un graphe de dépendance $G = (V, E)$ tel que $V$ est l'ensemble des objets du graphe et $E$ est l'ensemble des dépendances, une partition (cluster) est définie comme un sous-graphe $\acute{G} = (\acute{V}, \acute{E})$ dont les nœuds sont fortement reliés, i.e. cohésifs. Ces groupes sont considérés comme étant des implémentations de caractéristiques fonctionnelles. Nous supposons également que les implémentations de caractéristiques peuvent se chevaucher; un EdP peut être

[6] Les termes *fonctionnalité* et *caractéristique fonctionnelle* sont utilisées d'une manière interchangeable dans ce manuscrit

partagé par les implémentations de plusieurs caractéristiques simultanément. En outre, étant donné qu'une classe représente l'unité principale de construction dans les langages OO, nous supposons qu'une partition générée doit contenir au moins une classe. En effet, une classe est généralement considérée comme un ensemble de responsabilités qui simulent un concept ou une caractéristique dans le domaine d'application [Eyal-Salman, H. et al. 2013]. Cette hypothèse a été vérifiée par des expérimentations menées dans [Al-Msie'Deen, R. et al. 2013]. Par souci de simplicité, nous ne faisons aucune distinction entre une classe en soi et une classe abstraite. Compte tenu de cette hypothèse sur les classes, les interfaces doivent être écartées à partir de l'ensemble d'EdP. En effet, étant donné que le corps d'une méthode doit être de toute façon redéfini au niveau de la classe lors de l'implémentation d'une interface, l'ajout de ces interfaces à l'ensemble des données d'entrées ne va pas enrichir ce dernier par des informations supplémentaires et pertinentes. Par conséquent, l'élimination des interfaces à partir de l'ensemble initial d'EdP ne va pas affecter la validité de nos hypothèses.

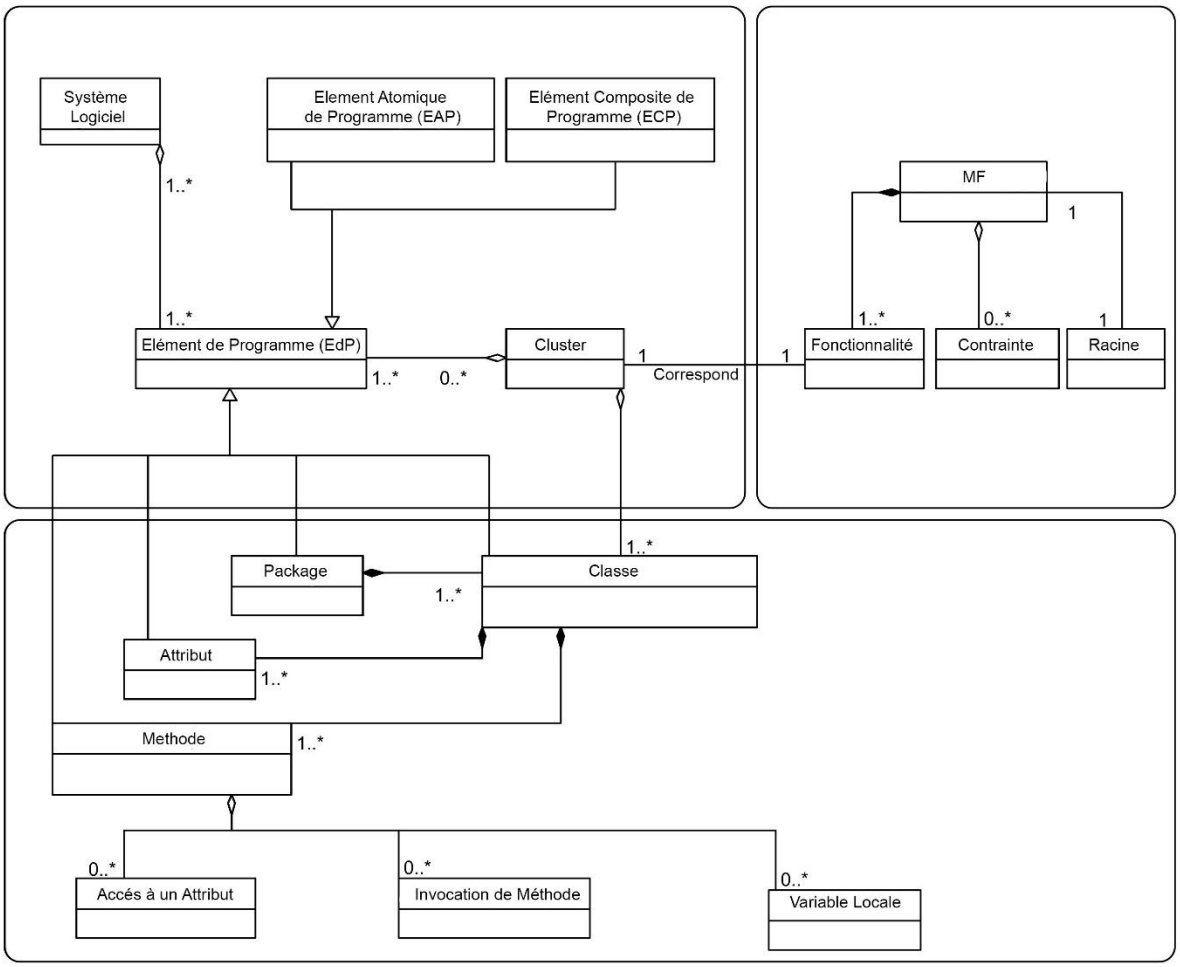



**Fig 4.1 : Un méta-model pour relier le code aux fonctionnalités.**

De plus, étant donné que les caractéristiques fonctionnelles d'un système logiciel sont associées à son comportement, nous avons décidé donc de ne garder que les EdP qui sont créés par le développeur pour implémenter des caractéristiques fonctionnelles spécifiques au système. Par exemple, une *LinkedList* est un concept, qui fait partie du domaine de solution, qui peut être implémenté dans le code, mais il n'est pas une caractéristique spécifique au système. Tous les concepts que nous avons définis pour l'extraction des fonctionnalités sont illustrés par un modèle de correspondances entre le programme et les fonctionnalités dans la figure *Fig 4.1*.

## 4.2 L'Extraction de fonctionnalités étapes par étapes

Cette section présente, de manière détaillée, le processus de découverte de fonctionnalités. Les données d'entrée ont été préparées selon la méthode décrite par Paskevicius et al. [Paskevicius, P. et al. 2012], tout en introduisant les changements nécessaires pour se conformer aux hypothèses et techniques utilisées dans notre approche. L'architecture de notre approche proposée pour l'extraction de fonctionnalités à partir du code est donnée dans la figure *Fig 4.2*.

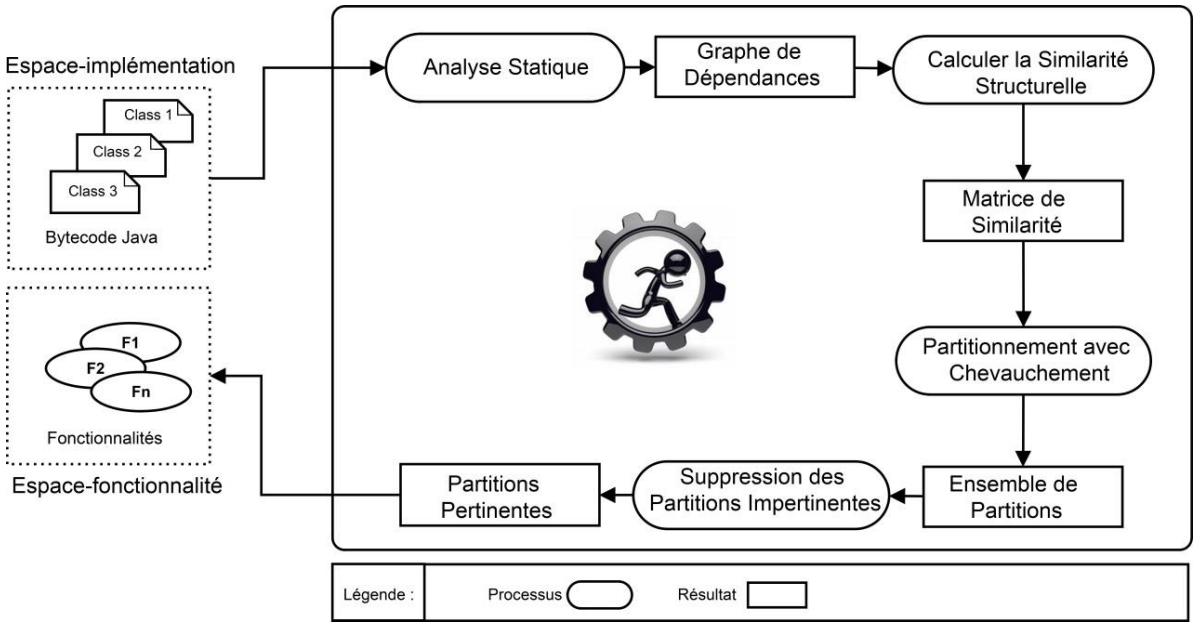



**Fig 4.2 : Processus de découverte de fonctionnalités.**

### 4.2.1 Extraction des EdP et des dépendances

Les dépendances du système candidat ont été modélisés en utilisant un graphe de dépendance orienté $G = (F, D)$, telle que $F$ est l'ensemble de sommets qui représentent les EdP, et $D$ est l'ensemble de dépendances. Le graphe de dépendance $G$ a été généré et sauvegardé dans un fichier XML en analysant les fichiers ".class" par l'outil *DependencyExtractor*. Ce dernier est une partie d'une

boîte à outils appelée *JDependencyFinder*[7]. L'utilisation du byte code Java au lieu du code source facilite l'analyse des systèmes existants dont le code source n'est pas disponible. En outre, le code source manque parfois des informations comme, par exemple, la façon dont le code compilé sera organisé dans des conteneurs d'exécution (fichiers JAR). Ces informations sont généralement définies dans les scripts exécutés lors de la compilation [Dietrich, J. et al. 2008]. Le choix d'utiliser un graphe de dépendance en entrée est justifié par la nature du problème, ainsi que la granularité des entités traitées. En effet, nous essayons de construire des regroupements d'EdP, i.e. des implémentations de fonctionnalités, en se basant sur les dépendances fonctionnelles entre ces EdP; une implémentation d'une fonctionnalité est caractérisée par une forte dépendance fonctionnelle (cohésion intra-cluster) entre ses EdP composants. DependencyExtractor a été exécuté via la ligne de commande en combinant trois de ses paramètres: [-class-filter], [-minimize] et [-filter-exclude]. Le paramètre [-minimize] a été utilisé pour supprimer les dépendances redondantes. En effet, il est souvent le cas qu'une dépendance explicite dans le code peut être déduite à partir d'une autre dépendance explicite dans ce code. Ces dépendances n'ajoutent rien à la connectivité globale du graphe et, par conséquent, doivent être enlevées. Le second paramètre, i.e. [-class-filtre], nous permet de sélectionner uniquement les dépendances entrantes/sortantes d'une classe. Cette dernière représente, comme expliqué précédemment, la principale unité de construction des implémentations de fonctionnalités. Le choix d'utiliser ce type de dépendances au lieu de ne considérer que les dépendances inter-classes est justifié par le fait que ce genre de dépendances mixtes sont susceptibles d'enrichir le jeu d'apprentissage. Enfin, le troisième paramètre [-filter-exclude] est utilisé pour supprimer les nœuds du graphe (resp. les dépendances) qui représentent des bibliothèques spécifiques au langage de programmation, telle que «Java.*» Et «javax.*».

### 4.2.2 Construire la matrice de similarité

Le graphe de dépendance généré dans l'étape précédente est utilisé à ce stade de travail pour construire une matrice de similarité. Une mesure simple basée sur la distance structurelle a été utilisée pour évaluer la similarité entre chaque paire de nœuds. Tout d'abord, le graphe de dépendance $G$ a été exprimé en tant qu'une matrice d'adjacence $C$ de taille$|F|$, de telle sorte que $c_{ij} = 1$ signifie qu'il existe une dépendance explicite de $i$ à $j$. Afin de décrire les dépendances indirectes, la matrice $C$ a été convertie en une matrice de distance $M$ de taille $|F|$ en utilisant l'algorithme des plus courts chemins entre toutes les paires de sommets de Floyd-Warshall (Floyd [Floyd, R.W. 1962] et Warshall [Warshall, S. 1962]), de telle sorte que $m_{ij}$ est égale à la distance du plus court chemin entre les deux éléments de programme $i$ et

[7] http://depfind.sourceforge.net/

$j$. Les deux matrices $C$ et $M$ sont asymétriques, car le graphe de dépendance $G$ est orienté. Étant donné que l'algorithme utilisé pour le partitionnement s'exécute sur un graphe non-orienté, la table asymétrique des distances $M$ a été convertie en une table symétrique $\acute{M}$, tel que $\acute{m}_{ij} = \acute{m}_{ji} = Min\,(m_{ij}, m_{ji})$. En outre, il est difficile d'estimer la similarité entre deux EdP en utilisant une distance absolue $\acute{m}_{ij}$. Par conséquent, la matrice des distances absolues a été convertie en une matrice normalisée de similarité de telle sorte que deux EdP ont une similarité $S\,(i,j) = 0$ s'il n'y a pas de chemin qui les relie, et une similarité $S\,(i,j) = 1$ s'ils sont identiques.

### 4.2.3 Construire les implémentations de fonctionnalités

Après la préparation des données d'apprentissage en utilisant les dépendances du programme, l'algorithme OclustR [Pérez-Suárez, A. et al. 2013] a été ensuite exécuté sur ces données pour générer un ensemble de partitions d'EdP. Chaque partition calculée est considérée comme l'implémentation d'une seule fonctionnalité. OclustR s'exécute en deux étapes principales: (1) l'étape d'initialisation, et (2) l'étape d'amélioration.

L'idée principale de la phase d'initialisation consiste à produire un premier ensemble $X$ de sous-graphes, i.e. *ws-graphes*, qui couvre le graphe; dans ce contexte, chaque ws-graphe consiste en une partition candidate. Ensuite, pendant la phase d'amélioration, un post-traitement est effectué sur les partitions initiales afin de réduire leur nombre et leur chevauchement. Pour ce faire, l'ensemble $X$ est analysé pour supprimer les ws-graphes qui sont considérés comme *moins utiles*. Ceux-ci sont élagués, en fusionnant les nœuds qui composent chacun d'entre eux avec les nœuds d'un ws-graphe choisis.

Formellement, soit $O = \{EdP_1,\ EdP_2\ ,\ \ldots,\ EdP_n\}$ un ensemble d'EdP. L'algorithme OClustR utilise comme entrée un graphe pondéré et non-orienté $\tilde{G}_\beta = (V, \tilde{E}_\beta, S)$, tel que $V = O$ et il existe une arête $(v,u) \in \tilde{E}_\beta$ ssi $v \neq u$ et $S(v,u) \geq \beta$, avec $S(EdP_1, EdP_2)$ est une fonction symétrique de similarité et $\beta \in [0,1]$ est un seuil défini par l'utilisateur ; chaque arête $(v,u) \in \tilde{E}_\beta$ est pondérée par la valeur de $S(v,u)$. Nous supposons que chaque EdP doit être affecté à une partition au moins, même si la similarité entre cet EdP et le centre de la partition est infime. Ainsi, il existe une arête $(v,u) \in \tilde{E}_\beta$ ssi $v \neq u$ et $S(v,u) > 0$. Par conséquent, nous sommes sûrs que chaque EdP dans le jeu d'apprentissage sera affecté au moins à une partition. L'algorithme OclustR n'a pas besoin, désormais, d'un paramètre d'entrée, ce qui augmente l'automatisation de la tâche. Toutefois, l'utilisateur peut toujours sélectionner d'autres valeurs pour le paramètre $\beta$ afin de générer des fonctionnalités avec une granularité plus fine, de telle sorte qu'il puisse avoir plusieurs vues du système analysé, ayant chacune un niveau d'abstraction différent.

### 4.2.4 Filtrage des résultats impertinents

Lors de la construction du graphe de dépendance, nous avons sélectionné uniquement les dépendances dans lesquelles les nœuds composant contiennent au moins une classe, ce qui signifie que les partitions résultantes seront composées d'EAP et/ou ECP. Tenant compte du fait que les classes sont considérées comme les principales unités de construction des implémentations de fonctionnalités, les partitions qui ne contiennent que des EAP sont considérées comme impertinentes et seront supprimées, par conséquent, de l'ensemble de résultats. Dans le cas d'une partition avec un contenu mixte, chaque EAP est remplacé par sa classe source. En effet, les EAP impliqués dans de telles partitions ne seront utilisés à ce stade du travail que pour fournir une vue optionnelle détaillée. Par conséquent, le résultat final est un ensemble de partitions pertinentes composées uniquement de classes. La figure *Fig 4.3* représente une fonctionnalité (i.e. un cluster) extraite par notre approche proposée pour le cas d'étude *drawing-Shapes* (voir la section 6.1.1). Étant donné qu'une partition calculée en utilisant OClustR est principalement un ws-graphe, elle est, par conséquent, définie par son nom (i.e. une référence unique donné par le système), son centre et ses satellites.

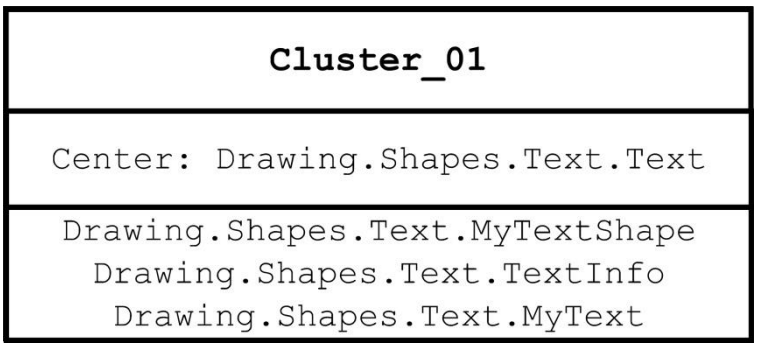



**Fig 4.3 : Exemple d'une fonctionnalité extraite automatiquement par notre approche.**

## 4.3 Evaluation de l'approche

Cette section présente les points forts qui caractérisent la contribution apportée par notre approche, ainsi que des points faibles qui limitent son utilisation dans certains contextes.

### 4.3.1 Avantages

L'approche extractive que nous avons présentée dans les sections précédentes pour la dérivation des implémentations de fonctionnalités des produits logiciels présente plusieurs avantages :

- **L'intelligibilité :** étant données que les fonctionnalités extraites sont constituées essentiellement d'EdP, cette nature formelle des résultats facilite alors leur manipulation ultérieure.

- **La réutilisabilité :** notre approche proposée permet d'extraire un catalogue de fonctionnalités au lieu de générer une seule hiérarchie. Cette caractéristique, en plus de la nature formelle des résultats, représente la force clé de notre approche proposée, de telle sorte qu'elle puisse fonctionner de manière générique et efficace. Cette généricité permet de rendre notre approche utile et réutilisable même en dehors des LdP. Les résultats obtenus par notre approche proposée peuvent être aussi manipulés, dans le contexte des LdP, par d'autres outils complémentaires afin d'obtenir des informations supplémentaires et, par conséquent, construire un MF fiable.
- **La flexibilité :** notre approche proposée adresse le problème de perte d'information, qui caractérise la plupart des méthodes basées sur ACF, par l'utilisation d'une mesure de similarité qui peut être ajustée par l'utilisateur. Cette flexibilité de paramétrage du seuil minimal de similarité entre les EdP donne donc aux utilisateurs la liberté de choisir entre plusieurs vues du système analysé en contrôlant la granularité des résultats.
- **La convenance** : par rapport à d'autres approches qui utilisent le partitionnement [Niu, N. and Easterbrook, S. 2008, Paskevicius, P. et al. 2012], notre approche proposée utilise un nouvel algorithme de partitionnement qui fournit des partitions qui se chevauchent d'une manière efficace. Ceci permet de prendre en considération le concept des préoccupations transversales.
- **L'automatisation :** l'implication de l'utilisateur est négligeable au cours de toutes les étapes du processus d'extraction. Par conséquent, la méthode peut être potentiellement très utile et peut réduire considérablement l'effort et le temps nécessaires pour dériver les fonctionnalités qui composent un système logiciel.

### 4.3.2 Limites

Notre approche de génération d'implémentations de fonctionnalités à partir du code a quelques limitations :

- Il y a une limitation à l'utilisation de l'algorithme de Floyd-Warshall pour inférer la similarité entre les EdP. En effet, la complexité en temps déterminé par cet algorithme est de $O(n^3)$ [Khuller, S. and Raghavachari, B. 2009]. En outre, pré-calculer tous les plus courts chemins et les stocker de manière explicite dans une grande matrice semble être difficile en termes d'espace. Donc, ces deux facteurs influent sur l'applicabilité de l'approche proposée sur les systèmes logiciels de tailles importantes. Même si le calcul des plus courts chemins est un problème bien étudié, les solutions exactes ne peuvent pas être adoptées pour un graphe massif de dépendance.

- Comme illustré ci-dessous (voir chapitre 6), bien que l'algorithme OClustR gère le chevauchement de partitions, il impose encore plusieurs restrictions lorsqu'il s'agit de partitions qui sont fortement chevauchée, ce qui limite l'utilisation de l'approche proposée pour les systèmes ayant un MF imbriqué.
- Un autre problème lié à OClustR se produit quand une classe donnée, i.e. une classe abstraite, est héritée par la plupart des classes du système. Cette classe sera donc considérée, lors de la phase d'initialisation de l'algorithme, comme étant un centre $c$ d'un ws-graphe $G_c^\star$ ayant comme satellites toutes les classes héritantes. Par conséquent, pendant la phase d'amélioration, chaque ws-graphe ayant comme centre l'un des satellites de $G_c^\star$ sera jugé comme impertinent. Dans ce cas, nous appelons $G_c^\star$ un *ws- graphe prédateur* et son centre $c$ un *centre prédateur*. Un tel phénomène peut donc affecter la précision des résultats.
- Enfin, la mesure basée sur la distance structurelle utilisée dans l'approche proposée impose encore quelques restrictions. En effet, nous avons utilisé une technique simple pour calculer la similarité entre les EdP en fonction du nombre de pas sur le plus court chemin qui les relie dans le graphe. Même si cette stratégie peut donner des résultats acceptables, elle ne tient pas compte de la multiplicité des chemins (i.e. la connectivité) entre une paire de nœuds.

## 4.4 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche pour la dérivation de fonctionnalités d'un produit logiciel. Cette approche synthétise automatiquement un catalogue d'implémentations de fonctionnalités spécifique à un produit Java à partir de byte code. Nous avons utilisé les dépendances qui existent entre les éléments de programme au niveau du code afin d'appliquer un algorithme de partitionnement de graphe d'une manière efficace.

L'approche proposée ci-dessus présente deux avantage : d'une part elle revisite le problème de la synthèse des fonctionnalités dans le contexte du rétro-ingénierie des fonctionnalités en général, et d'autre part elle représente un appui fort pour mener un processus de rétro-ingénierie d'une LdP. A ce propos, le chapitre suivant illustre l'utilité de cette approche dans le contexte des LdP.

# Chapitre 5: EXTRACTION DE LA LIGNE DE PRODUITS

Ce chapitre présente la deuxième contribution de notre approche qui consiste à la configuration d'un ensemble de variantes logicielles en une LdP en se basant sur notre approche présentée dans le chapitre précédent. La section 5.1 présente l'idée globale derrière cette contribution. La section 5.2 explique, d'une manière détaillée, les différentes étapes du processus d'extraction de la LdP. La section 5.3 explique les points forts et les points faibles de cette approche proposée. Enfin, la section 5.4 conclut ce chapitre.

## 5.1 Objectif et hypothèses

L'objectif général de notre travail est d'identifier les commonalités et les variabilités pour une collection de variantes logicielles en se basant sur la sémantique de leurs codes. Notre approche, contrairement aux approches existantes, prend en charge l'extraction de commonalités et de variabilités à partir de différentes applications qui sont développées par différents programmeurs dans différentes organisations, utilisant plusieurs langages de programmation, tout en adressant le même domaine d'application. Bref, l'avantage principal de cette approche est la possibilité de défier le problème d'hétérogénéité entre les variantes logicielles en se basant sur la sémantique cachée derrière leur code source. Par souci de simplicité, nous nous limitons dans le cadre de ce travail aux variantes logicielles qui sont programmées avec le langage Java. Plus précisément, nous utilisons leurs code objet Java.

Pour extraire une LdP à partir d'un ensemble de variantes logicielles nous utilisons, comme point de départ, notre approche présentée dans le chapitre précédent sur l'extraction de fonctionnalités d'un système existant. En effet, pour extraire une LdP à partir d'un ensemble de variantes logicielles qui sont hétérogènes au niveau du vocabulaire utilisé, nous supposons que la façon la plus sûre est d'extraire, en premier lieu, un catalogue de fonctionnalités pour chacune de ces variantes. Ensuite, les fonctionnalités de chaque variante doivent être annotées par les concepts les plus discriminants du domaine. Ces concepts sont extraits à partir des noms d'EdP trouvés dans le code. Nous supposons aussi que les noms d'EdP sont composés d'un ou plusieurs mots naturels provenant de la langue anglaise. Cette hypothèse est justifiée par les mécanismes de fonctionnement des techniques et outils de traitement de langage naturel qui sont utilisées dans notre approche. Aussi, et par soucis de simplicité, nous ne considérons pas l'expansion d'acronymes et d'abréviations qui peuvent être trouvés dans les noms d'EdP.

En représentant chaque fonctionnalité par un vecteur de concepts normalisés qui décrivent leurs contenus, nous espérons donc calculer la similarité sémantique entre ces fonctionnalités. Nous supposons aussi que l'ontologie WordNet est appropriée pour notre travail vu que la relation d'hypernymie/hyponymie « est-un»

est la plus dominante entre les concepts et qu'elle représente 80% des relations dans WordNet. Cet hypothèse est soutenu par la plupart des approches qui ont essayé à tirer profit de l'aspect sémantique dans un domaine donné en utilisant WordNet, et qui ont approuvé la richesse de cet ontologie en terme de contenue informationnel [Rada, R. et al. 1989, Desmontils, E. and Jacquin, C. 2002, Seco, N. et al. 2004].

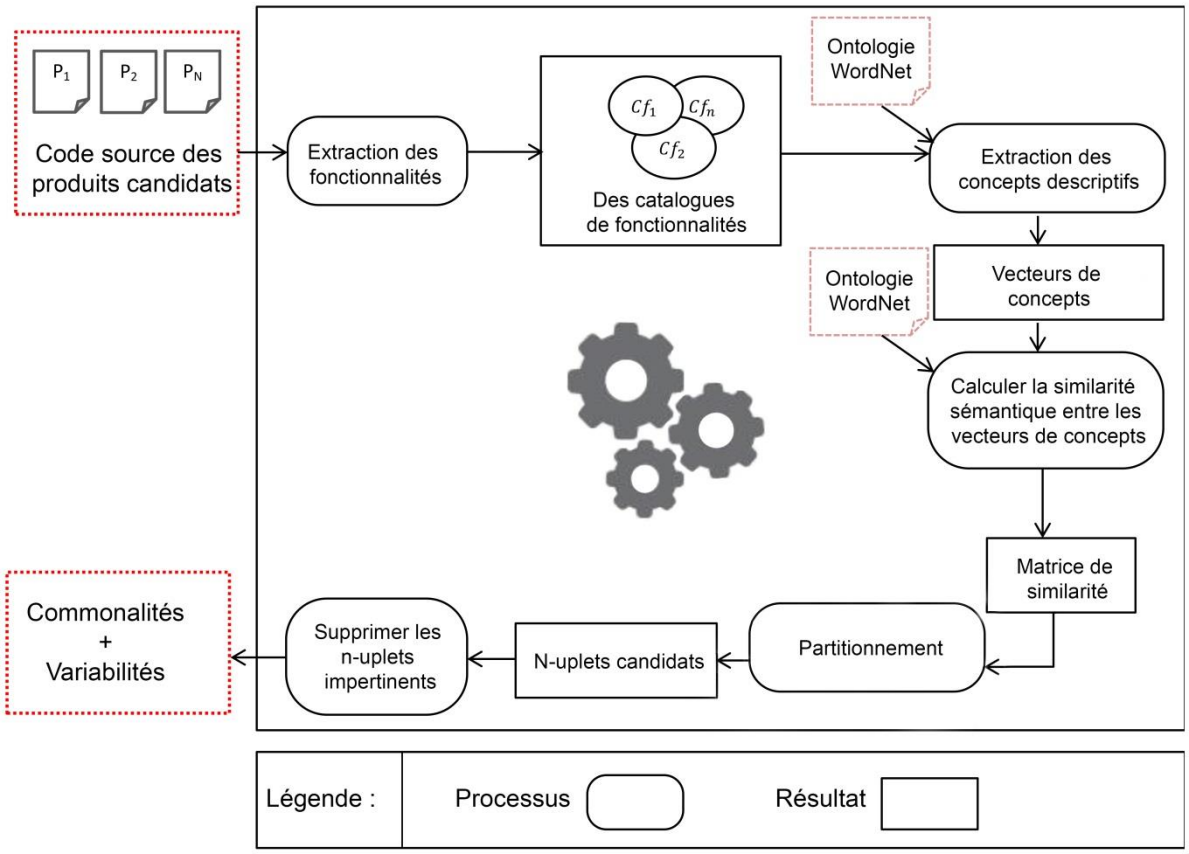



**Fig 5.1 : Extraction de commonalités et de variabilités en utilisant les ontologies.**

Calculer la similarité inter-fonctionnalités se résume désormais à comparer tous les vecteurs de concepts, deux à deux, utilisant une stratégie indirecte. Pour ce faire nous calculons d'abord la similarité entre les paires de concepts utilisant une mesure de similarité entre paires de concepts. Ensuite nous utilisons une technique d'agrégation pour agréger les résultats obtenus pour les concepts de chaque vecteur. Nous obtenons donc une matrice symétrique $S$ de similarité de dimensions $n \times n$, tel que $n$ est le nombre total de fonctionnalités de l'ensemble des variantes $P_i$, et chaque entrée $s_{xy}$ dans la matrice $S$ représente le degré de similarité entre une fonctionnalité $x$ et une fonctionnalité $y$. A partir de cette matrice de similarité, nous pouvons alors décider quelles sont les fonctionnalités identiques qui existent dans toutes les variantes, i.e. les commonalités, et quelles sont les fonctionnalités identiques qui sont partagées seulement par quelques variantes, i.e. les variabilités.

Deux fonctionnalités sont jugées comme identiques si leur similarité est supérieure ou égale à un seuil fixé par l'analyste. La figure *Fig 5.1* illustre ce processus d'extraction de LdP. Le processus commence utilisant le code des variantes logicielles comme état initial et se termine par la production de la liste complète de commonalités et de variabilités de la LdP.

## 5.2 Extraction d'une LdP étape par étape

Cette section présente, de manière détaillée, le processus d'extraction d'une LdP à partir d'un ensemble de variantes logicielles Java.

### 5.2.1 Extraction de concepts descriptifs

Après avoir utilisé l'approche proposée dans le chapitre précédent pour extraire les fonctionnalités de chacune des variantes logicielles $P_i$, nous obtenons donc pour chacune d'entre elles un vecteur de fonctionnalité de taille $k$ dénoté $Cf_i$, t.q $Cf_i = \{f_{i,1}, f_{i,2}, \dots, f_{i,k}\}$. Nous utilisons à cette étape une vue détaillée de chaque fonctionnalité dérivée (i.e. son contenu brut). En effet, les fonctionnalités utilisées ici sont composées d'EAP et d'ECP et non seulement d'ECP. Cela est susceptible donc d'enrichir les données en entrée du processus d'extraction des concepts. Ce choix n'est pas contradictoire avec nos hypothèses car les concepts extraits à partir des ECP vont avoir une pondération plus élevée par rapport à ceux qui sont dérivés à partir des EAP. L'utilisation des EAP permettra plutôt de former un contexte qui est nécessaire pour désambiguïser les sens des concepts extraits dans les étapes ultérieures (voir section 5.2.1.2).

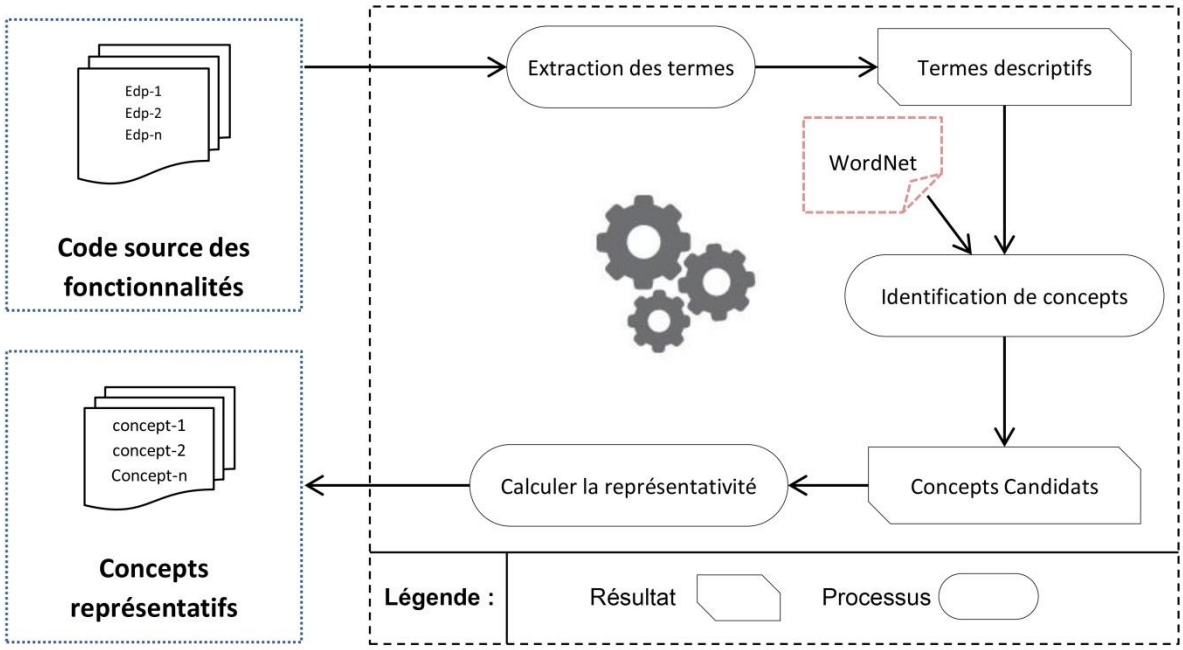



**Fig 5.2 : Processus d'extraction des concepts représentatifs.**

Les vecteurs de fonctionnalités obtenus dans l'étape précédente constituent, à ce stade de travail, des données d'entrées. Pour extraire les concepts descriptifs à partir des implémentations de fonctionnalités, nous utilisons les noms d'identificateurs dans le code. En effet, les noms d'EdP jouent un rôle indispensable et sont l'un des moyens utilisés par les programmeurs pour conserver et communiquer les intentions ou les noms des concepts du monde réel qui sont implémentés à travers les EdP [Rajlich, V. 2009]. Les noms d'identificateurs constituent approximativement 70% du code source [Deissenbock, F. and Pizka, M. 2005]. L'extraction des concepts doit passer par trois étapes principales, allant de (1) la construction d'une liste de termes, (2) l'extraction des concepts candidats, jusqu'à (3) la suppression des concepts impertinents. Le processus simplifié d'extraction de concepts descriptifs est illustré dans la figure *Fig 5.2*.

#### 5.2.1.1 Construire une liste de termes

Les programmeurs orientés objet suivent habituellement des conventions de nommage largement connues et couramment adoptées. Ils utilisent les mêmes règles grammaticales (souvent implicites) pour nommer les éléments de programme qui sont structurellement identiques. Ces règles indiquent le rôle de chaque terme dans un nom d'EdP. Par exemple, les noms de méthodes sont souvent construits à partir de verbes, qui peuvent être suivis de noms, tandis que les noms de classe sont souvent des séquences de noms. Les rôles des termes peuvent être exploités pour en extraire des concepts. Afin de construire la liste des termes nous nous inspirons de l'approche proposée par Abebe et al. [Abebe, S.L. et al. 2013, Abebe, S.L. and Tonella, P. 2015]. La démarche que nous avons suivie pour extraire la liste des termes est illustrée dans la figure *Fig 5.3*.

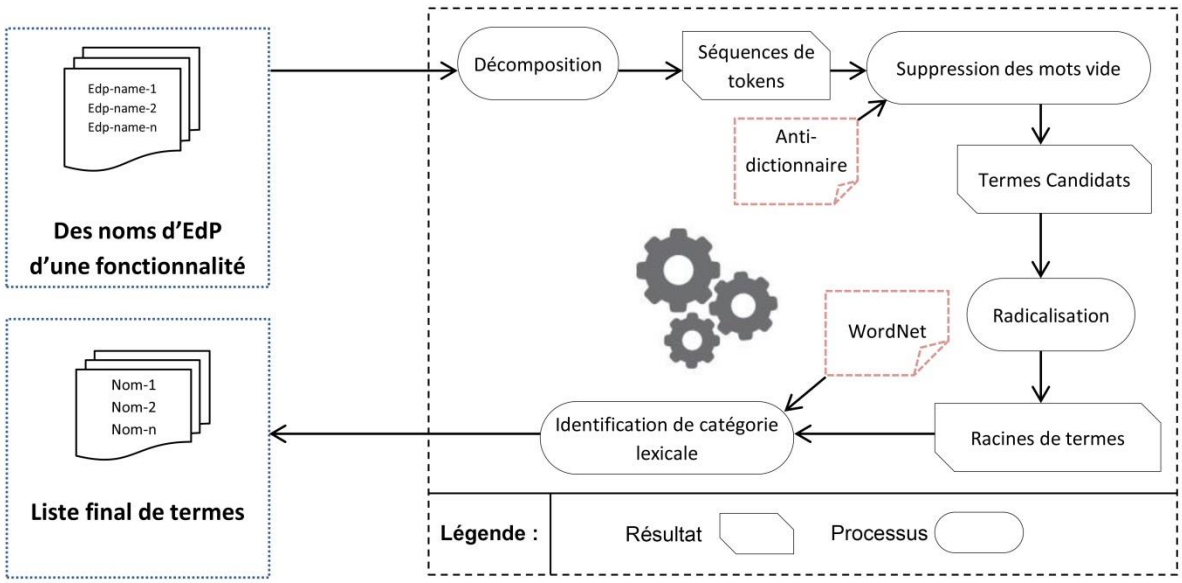



**Fig 5.3 : Processus d'extraction des termes descriptifs.**

#### 5.2.1.1.1 La tokenisation

Dans un premier temps, les noms d'EdP (classes, attributs, et méthodes) sont d'abord obtenus à partir du code des fonctionnalités qui sont précédemment construites utilisant notre approche décrite dans le chapitre précédent. Dans le cas où un nom d'EdP est composé de plusieurs termes, nous procédons à un processus de décomposition (i.e. anglicisme *tokanization*) pour extraire les entités lexicales qui les composent (i.e. *tokens*). Pour ce faire, nous tirons profit des séparateurs couramment utilisés, tels que les *CamelCase* (ex. *CopyPhoto*) et les *underscores* (ex. *copy_photo*) et les caractères numériques (ex. *code2text*). Ceci peut être réalisé également en utilisant des techniques plus sophistiquées dont un comparatif de ces techniques a été proposé dans [Hill, E. et al. 2014]. Avant de diviser un nom d'EdP en une liste de termes qui le composent, des préfixes, tels que ceux liés à la notation hongroise [Simonyi, C. 1999] (par exemple, `s_` pour les variables statiques, et `l_` pour les variables locales) sont enlevés.

De plus, étant donné que nous analysons des EdP extraits du byte code Java, nous devons donc considérer un autre type de nommage spécifique aux EdP dits « *synthétiques* ». Un EdP synthétique est un EdP créé dynamiquement par le compilateur, et qui ne peut pas être visible directement dans le code source Java. Un exemple d'EdP synthétique est représenté par le cas des classes internes (*inner classes*) et les classes anonymes (*anonymous classes*), où une classe synthétique est créée par le compilateur pour ces types de classes. Les noms d'EdP synthétiques sont composés, dans l'exemple d'une classe interne ainsi que ses attributs et méthodes, de la signature de l'EdP englobant (i.e. classe) suivi du caractère spécial «$ » et le nom originalement attribué à la classe interne dans le code source. Nous ne pouvons pas donc négliger ce type d'EdP vu qu'il contient de l'information supplémentaire et pertinente dans leurs noms. Nous avons exploré les différents formats possibles de nommage pour différents types d'EdP synthétiques afin de découvrir les patrons de nommage utilisés par le compilateur. Une petite liste prédéfinie des différentes formes possibles a été donc utilisée pour créer des filtres afin d'extraire les termes pertinents qui sont enfermés dans les noms d'EdP synthétiques.

#### 5.2.1.1.2 Suppression des mots vides

Après avoir extrait la liste des termes qui composent chaque EdP, nous procédons à la suppression des mots vides (i.e. stop words) utilisant un anti-dictionnaire. Les mots vides sont des mots qui sont tellement communs qu'il est inutile de les considérer, tel que « the », « a », « by », « all », …etc. La suppression des mots vides à ce stade de travail élimine l'information impertinente dès le début et améliorera, par conséquent, la performance du système durant les étapes ultérieures du processus d'extraction de concepts en réduisant l'espace de recherche. Etant donné

que les mots vides se réfèrent habituellement aux mots les plus courants d'une langue, il n'existe pas donc une liste unique et universelle de mots vides qui est utilisée par tous les outils de traitement du langage naturel. Vu que nos traitements ultérieurs sont principalement basés sur WordNet, nous avons donc choisi un dictionnaire de 199 mots vides fournit avec le package *WordNet-InfoContent-3.0* qui est spécifique à l'outil *WordNet::Similarity*[8] développé par Ted Pedersen.

#### 5.2.1.1.3 La radicalisation

Avant d'utiliser un terme pour identifier le concept (ou plusieurs concepts) ontologique qui lui correspond, et pour éviter des représentations différentes de ce terme en raison de flexions d'un mot, par exemple «*connections*», «*connect*», «*connected*» et «*connecting*», nous désuffixons les termes en utilisant l'algorithme *Porter Stemmer* [Porter, M.F. 1980]. En effet, de tels termes avec une racine commune ont généralement des significations semblables. Le processus de radicalisation des mots utilisant des outils automatiques est une opération qui est particulièrement utile dans le domaine de la recherche d'information. Généralement, la performance d'un système RI est améliorée lorsque les groupes de termes avec une racine commune sont confondus en un seul terme. Cela peut être fait par l'élimination des différents suffixes, *-ed*, *-ing*, *-ion*, *-ions*, afin de ne garder que la racine commune (dans l'exemple précèdent la racine est : «*connect*»). Le processus de radicalisation permet de réduire le nombre total de termes dans le système de RI et, par conséquent, réduit la taille et la complexité des données dans le système, ce qui est toujours rentable.

La force clé de l'algorithme Porter Stemmer, qui a motivé son utilisation dans cette étape de notre approche, réside dans le fait qu'il n'utilise pas un dictionnaire exhaustif de racines et qu'il est très approprié pour améliorer une tache de RI. Les suffixe sont simplement enlevés pour améliorer la performance d'un système de RI. Au lieu d'utiliser un dictionnaire, l'algorithme utilise seulement une petite liste prédéfinie des différentes formes possibles de suffixes, et avec chacun des suffixes le critère selon lequel ce dernier peut être éliminé afin de former une racine valide. Les principaux avantages de l'implémentation de cet algorithme est qu'elle est de petite taille (environ 200 LOC), elle s'exécute rapidement, elle est raisonnablement simple, facile à comprendre et publiquement disponible dans la page officielle[9] de l'auteur en différents langages de programmation. Les expérimentations menées par les auteurs de Porter Stemmer ont montré qu'il est capable d'améliorer la performance d'un système de RI, en le comparant à un autre système plus sophistiqué et longuement utilisé dans ce domaine [Porter, M.F. 1980].

[8] http://wn-similarity.sourceforge.net/
[9] http://tartarus.org/martin/PorterStemmer/

#### 5.2.1.1.4 Identification de catégorie lexicale

Il est généralement affirmé que la sémantique d'un langage est surtout capturée par des noms ou des phrases nominales. Ainsi, les concepts capturés dans notre approche proviennent principalement des noms trouvés dans les listes de termes qui sont construites dans l'étape précédente. Par soucis de simplicité, nous ne considérons que les concepts représentés par un seul terme (i.e. un nom). L'identification des éventuelles catégories lexicales d'un terme, afin de ne choisir que des noms, est basée sur l'utilisation de WordNet. En effet, WordNet répertorie toutes les catégories lexicales possibles pour un terme. Afin de tirer profit de l'information fournit par l'ontologie WordNet, nous utilisons les fichiers de sa base données (version 3.0[10]) en combinaison avec la bibliothèque *Java WordNet Interface* JWI[11] (version 2.2.4) développée par Mark Alan Finlayson [Finlayson, M.A. 2014]. Le choix d'utiliser JWI est justifié par résultats des expérimentations menées par son auteur et qui ont démontré qu'elle est plus performante et simple à utiliser par rapport à d'autres API similaires [Finlayson, M.A. 2014]. Nous considérons la catégorie d'un terme comme nominale si au moins l'un des concepts (i.e. synset) correspondant à ce terme dans l'ontologie WordNet est un nom.

### 5.2.1.2 Identification des concepts et instances

Dans le code d'un logiciel, des termes qui se réfèrent à des concepts du domaine sont utilisés avec d'autres termes afin d'exprimer des connaissances du domaine. Ce mixage de termes référant à des concepts du domaine en question avec d'autres termes auxiliaires est inévitable. Un concept qui est représenté par un terme dans le code est considéré comme pertinent en fonction du contexte dans lequel il est utilisé. Par exemple, un concept clé qui est pertinent à une caractéristique fonctionnelle d'un système informatique peut être jugé comme non pertinent s'il est utilisé dans un contexte qui adresse des caractéristiques non fonctionnelles de ce système. De plus, certains termes existent dans la majorité des noms d'EdP, due aux conventions de nommage dans le contexte de la programmation orientée objet, et peuvent être considérés comme des mots vides qui ne reflètent aucun concept pertinent.

Dans leur approche, Desmontils et al. [Desmontils, E. and Jacquin, C. 2002] ont bien étudié cette notion de contexte en évaluant l'importance d'un concept extrait à partir d'un terme dans une page HTML selon trois facteurs principaux :

- Le nombre d'occurrences du terme représentant le concept en question ;
- L'importance du terme indiquée par son emplacement dans le document, et qui est relative à la balise HTML qui l'entoure ;

[10] https://wordnet.princeton.edu/wordnet/download/current-version/
[11] http://projects.csail.mit.edu/jwi/

- Les relations du concept en question avec d'autres concepts de la même page.

Nous nous inspirons donc du travail de Desmontils afin d'identifier les concepts et les instances dans le code utilisant la liste des termes construite dans l'étape précédente. Nous calculons d'abord le poids de fréquence $P\_Freq(T_i)$ pour chaque terme $T_i$ en fonction de son nombre d'occurrence dans le code de la fonctionnalité, ainsi que le type de l'EdP duquel il a été extrait. Le type de l'EdP dans notre contexte remplace la balise HTML. Le coefficient correspondant à chaque type d'EdP doit être donc fixé expérimentalement. L'outil *JDependencyFinder* utilisé dans notre approche d'extraction de fonctionnalité (voir chapitre 4) distingue trois types d'EdP : (1) les package, (2) les classes, et (3) les attributs et méthodes. Etant donné qu'une classe est considérée comme la principale unité de construction d'implémentations de fonctionnalités et qu'elle simule un concept ou une caractéristique dans le domaine d'application [Eyal-Salman, H. et al. 2013], une première approche simple consiste donc à choisir un coefficient fort pour les concepts extraits à partir des noms de classes et un coefficient qui est moins fort pour les concepts qui sont extraits à partir des noms de variables ou de méthodes. En plus, nous ne faisons aucune distinction entre les EdP normaux et les EdP synthétiques.

Soit un terme $T_i$ qui apparait $p$ fois dans le code d'une fonctionnalité contenant $n$ termes ($i \in [1, n]$), et $M_{i,j}$ le coefficient du type de l'EdP dont le nom est la source de l'occurrence $j$ du terme $T_i$. Le poids de fréquence $P\_Freq(T_i)$ est donné par :

$$P_{Freq(T_i)} = \frac{P(T_i)}{max_{k=1..n}\big(P(T_k)\big)}; \quad P(T_i) = \sum_{i=1}^{p} M_{i,j} \tag{26}$$

Dans une deuxième étape, nous calculons la *similarité cumulative* ainsi que *le coefficient de représentativité* utilisant les mêmes formules proposées par Desmontils, et qui ont été déjà expliquées plus haut dans le chapitre 2 de ce document (voir section 2.3.2.2). La similarité cumulative pour un synset est calculée utilisant la formule suivante :

$$\widehat{sim}\big(synset_i(T_k)\big) = \sum_{j \in [1,k-1] \cup [k+1,m]} \sum_{l=1}^{l_j} sim\Big(synset_i(T_k), synset_l\big(T_j\big)\Big) \tag{27}$$

Tel que $sim(synset_i(T_k), synset_l(T_j))$ représente la similarité sémantique entre deux synsets. Pour calculer la similarité sémantique pour une paire de synsets, Desmontils [Desmontils, E. and Jacquin, C. 2002] utilise dans son approche la mesure de similarité structurelle de Wu & Palmer [Wu, Z. and Palmer, M. 1994]. Nous utilisons dans le cadre de notre travail la mesure à base du contenu informationnel proposée par Lin [Lin, D. 1998]. (voir Section 2.3.3.2)

$$Sim_{Lin}(c_1, c_2) = \frac{2 * \mathrm{Sim}_{Resnik}(c_1, c_2)}{CI_{Resnik}(c_1) + CI_{Resnik}(c_2)} \; ; \; Sim_{Lin} \in [0..1] \tag{28}$$

$$\mathrm{Sim}_{Resnik}(c_1, c_2) = CI\big(lso(c_1, c_2)\big) \tag{29}$$

$$CI_{Resnik}(c) = -\log p(c) \tag{30}$$

Le choix de cette mesure est un compromis entre deux critères : (1) la précision de cette mesure par rapport à d'autres mesures existantes, et (2) la disponibilité d'un API open source qui implémente cette mesure. En effet, cette mesure s'est montrée plus performante en terme de précision selon plusieurs études comparatives [Lin, D. 1998, Slimani, T. 2013], avec un taux te précision d'environ 0.82. De plus, une implémentation de cette mesure est fournie dans la bibliothèque *Java Wordnet::Similarity library* (JWS). La bibliothèque JWS[12] développée par David Hope en 2008 à l'université de Sussex est une adaptation Java créée à partir de la bibliothèque *Wordnet::Similarity*[13] originalement créée en langage Perl par Ted Pedersen. Elle fournit un certain nombre de mesures de similarité sémantique et de proximité sémantique basées sur WordNet. Etant donné deux synsets, elle calcule un score numérique montrant leur degré de similarité. JWS utilise des fichiers de contenus informationnels pré-calculés à partir de plusieurs corpus, tels que : «*British National Corpus BNC*» (World Edition), «*The Brown Corpus*» qui a été utilisé par Resnik [Resnik, P. 1995] et Lin [Lin, D. 1998] dans leurs expérimentations, et «*SemCor*». Nous avons choisi d'utilisé la configuration par défaut de JWS qui utilise SemCor.

Ensuite, le coefficient de représentativité qui indique le pouvoir discriminant d'un synset dans une fonctionnalité est calculé pour chaque synset candidat utilisant la formule suivante :

$$Rep\big(synset_i(T_k)\big) = \frac{\alpha * P_{Freq(synset_i(T_k))} + \beta * \widehat{sim_{rescaled}}\big(synset_i(T_k)\big)}{\alpha + \beta} \tag{31}$$

$\widehat{sim_{rescaled}}\big(synset_i(T_k)\big)$ représente la valeur normalisée de la similarité cumulative $\widehat{sim}\big(synset_i(T_k)\big)$, tel que $\widehat{sim_{rescaled}}\big(synset_i(T_k)\big) \in [0,1]$. Cette valeur normalisée a été calculée utilisant la formule suivante :

$$\widehat{sim_{rescaled}}\big(synset_i(T_k)\big) = \frac{\widehat{sim}\big(synset_i(T_k)\big) - \mathrm{minSc}}{\mathrm{minSc} + \mathrm{maxSc}} \tag{32}$$

Tel que minSc et maxSc représentent respectivement la valeur minimale et la valeur maximale de la similarité cumulative des Sysnset d'une fonctionnalité. Cette étape de normalisation garantie d'avoir une valeur normalisée du coefficient de représentativité $Rep\big(synset_i(T_k)\big)$.

[12] JWS n'est plus disponible sur le net, donc nous avons dû contacter son auteur David Hope pour obtenir une copie.

[13] http://wn-similarity.sourceforge.net/

De plus, nous utilisons les valeurs par défaut pour $\alpha$ et $\beta$ qui ont été fixés expérimentalement par Desmontils à 1 et 2 respectivement [Desmontils, E. and Jacquin, C. 2002]. Nous utilisons aussi un seuil minimal de représentativité $minRep$ afin de ne garder que des Synsets pertinents.

### 5.2.2 Calculer la similarité des fonctionnalités

A la fin de l'étape précédente nous obtenons donc, pour chacune des variantes logicielles $P_i$, un catalogue de fonctionnalités $Cf_i$, t.q $Cf_i = \{f_{i,1}, f_{i,2}, \dots, f_{i,k}\}$. Chaque fonctionnalité $f_{i,j}$ est annotée par un ensemble de concepts (i.e. synsets) $C_{i,j} = \{c_1, c_2, \dots, c_t\}$. A cette étape du processus d'extraction nous devons calculer la similarité sémantique entre les fonctionnalités de toutes les variantes logicielles. Pour ce faire nous construisons d'abord un ensemble de tous les couples possibles de fonctionnalités. Cet ensemble est obtenu à partir du produit cartésien des catalogues de fonctionnalités des variantes $P_i$. Pour chaque deux variantes logicielles différentes $P_i$ et $P_j$, nous obtenons alors des couples de la forme ($f_{i,x}$, $f_{j,y}$) t.q. $f_{i,x} \in Cf_i$ et $f_{j,y} \in Cf_j$ et $i \neq j$. Le calcul de similarité sémantique pour une paire de fonctionnalités se résume désormais à calculer la similarité sémantique entre les deux vecteurs de concepts associés aux composantes de cette paire.

Le calcul de similarité entre deux vecteurs de concepts peut être effectué utilisant l'une des techniques expliquées précédemment (voir section 2.3.3). Nous utilisons dans le cadre de notre travail une stratégie indirecte pour comparer les vecteurs de concepts comme suit :

1. Calculer les similarités des paires de concepts obtenues à partir du produit cartésien des deux vecteurs utilisant la mesure de Lin [Lin, D. 1998] implémentée dans la bibliothèque JWS.
2. Agréger les résultats obtenus dans l'étape « 1 » utilisant la technique du meilleur résultat avec similarité moyenne BMA (voir section 2.3.3.3).

Après avoir calculé les similarités pour toutes les paires de concepts qui décrivent deux fonctionnalités $U$ et $V$, nous pouvons donc utiliser une méthode d'agrégation pour généraliser sur ces valeurs de similarités et obtenir la similarité entre les fonctionnalités qui leurs sont associées. Pour ce faire nous utilisons la technique du meilleur résultat avec similarité moyenne BMA (voir section 2.3.3.3).

$$Sim_{BMA}(U,V) = \frac{AVG_u\big(MAX_v sim(u,v)\big) + AVG_v\big(MAX_u sim(u,v)\big)}{2} \; ; \; u \in U, v \in V \quad (33)$$

### 5.2.3 Identification de variabilités et de commonalités

Après avoir calculé la similarité pour les paires de fonctionnalités des variantes logicielles, nous obtenons donc une matrice symétrique $S$ de similarité t.q. $|S| =$

$m \times m$ et $m$ est le nombre totales des fonctionnalités de l'ensemble $V$ des variantes logicielles, et chaque entrée $s_{xy}$ dans la matrice $S$ représente le degré de similarité entre une fonctionnalité $x$ et une fonctionnalité $y$ et $s_{xy} \in [0,1]$. Nous rappelons que les noms des fonctionnalités extraites utilisant notre approche d'extraction présentée précédemment sont des identifiants uniques qui ont été choisis automatiquement par le système, et qui sont de la forme $f_{i,j}$ qui signifie la $j^{\text{éme}}$ fonctionnalité de la variante logicielle $P_i$. Par souci de simplicité, la documentation (i.e. renommage) des fonctionnalités extraites ne fait pas partie de notre travail actuel. Le tableau *Tab 5.1* montre un exemple illustratif de la table $S$. Nous avons utilisé trois variantes logicielles dans cet exemple :

- La variante $P_1$, t.q. $Cf_1 = \{f_{1,1}, f_{1,2}, f_{1,3}, f_{1,4}\}$
- La variante $P_2$, t.q. $Cf_2 = \{f_{2,1}, f_{2,2}, f_{2,3}\}$
- La variante $P_3$, t.q. $Cf_3 = \{f_{3,1}, f_{3,2}, f_{3,3}\}$
- $m = |V| = |Cf_1 \cup Cf_2 \cup Cf_3| = 10$

| | $f_{1,1}$ | $f_{1,2}$ | $f_{1,3}$ | $f_{1,4}$ | $f_{2,1}$ | $f_{2,2}$ | $f_{2,3}$ | $f_{3,1}$ | $f_{3,2}$ | $f_{3,3}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $f_{1,1}$ | 1 | | | | 0.46 | 0.44 | 0.41 | 0.69 | 0.23 | 0.24 |
| $f_{1,2}$ | | 1 | | | 0.24 | 0.92 | 0.43 | 0.29 | 0.41 | 0.64 |
| $f_{1,3}$ | | | 1 | | 0.26 | 0.34 | 0.78 | 0.19 | 0.29 | 0.37 |
| $f_{1,4}$ | | | | 1 | 0.33 | 0.44 | 0.32 | 0.26 | 0.69 | 0.44 |
| $f_{2,1}$ | 0.46 | 0.24 | 0.26 | 0.33 | 1 | | | 0.57 | 0.43 | 0.26 |
| $f_{2,2}$ | 0.44 | 0.92 | 0.34 | 0.44 | | 1 | | 0.36 | 0.40 | 0.63 |
| $f_{2,3}$ | 0.41 | 0.43 | 0.78 | 0.32 | | | 1 | 0.34 | 0.29 | 0.36 |
| $f_{3,1}$ | 0.69 | 0.29 | 0.19 | 0.26 | 0.57 | 0.36 | 0.34 | 1 | | |
| $f_{3,2}$ | 0.23 | 0.41 | 0.29 | 0.69 | 0.43 | 0.40 | 0.29 | | 1 | |
| $f_{3,3}$ | 0.24 | 0.64 | 0.37 | 0.44 | 0.26 | 0.63 | 0.36 | | | 1 |

**Tab 5.1 : Exemple d'une table $S$ des similarités entre les fonctionnalités extraites à partir des variantes logicielles.**

Les cellules grisées ne contiennent pas de valeurs vu que nous ne sommes pas en train de calculer la similarité entre les fonctionnalités de la même variante ; chaque fonctionnalité d'une variante $P_i$ est unique par rapport aux autres fonctionnalités dans le même catalogue de fonctionnalité $Cf_i$. La valeur de similarité entre chaque fonctionnalité et elle-même est maximale ($s = 1$).

Afin d'identifier les variabilités et les commonalités des variantes logicielles analysées, nous avons besoins d'abord de fonctions et de notions auxiliaires qui sont définies comme suit :

$\boldsymbol{mostSimilarFeature(f_{x,i}, P_y)}$ **:** Soient $P_{\mathrm{x}}$ et $P_{\mathrm{y}}$ deux variantes logicielles, avec deux catalogues de fonctionnalité $Cf_{\mathrm{x}}$ et $Cf_{\mathrm{y}}$, respectivement. La fonction $mostSimilarFeature(f_{x,i}\ ,\ P_y)$ calcule la fonctionnalité la plus similaire, dans le catalogue $Cf_{\mathrm{y}}$ de $P_y$, par rapport à une fonctionnalité $f_{x,i}$ de $P_{\mathrm{x}}$. Dans l'exemple précédent, on trouve que $mostSimilarFeature(f_{1,1}, P_2) = f_{2,1}$, parce que $S(f_{1,1}, f_{2,1}) = 0.46$ qui est supérieur à $S(f_{1,1}, f_{2,2}) = 0.44$ et supérieur à $S(f_{1,1}, f_{2,3}) = 0.41$

$\boldsymbol{isIdenticalFeature(f_{x,i}, f_{y,j})}$ **:** Une fonction booléenne qui vérifie que ses deux paramètres sont deux implémentations différentes de la même fonctionnalité dans deux variantes logicielles distinctes. La fonction $isIdenticalFeature()$ renvoie une valeur «$vrai$» pour les deux fonctionnalités $f_{x,i}$ et $f_{y,j}$ appartenant respectivement aux deux variantes $P_{\mathrm{x}}$ et $P_{\mathrm{y}}$, ssi :

1. $S(f_{x,i}, f_{y,j}) \geq \Theta$
2. $mostSimilarFeature(f_{x,i}, P_y) = f_{y,j}$
3. $mostSimilarFeature(f_{y,j}, P_x) = f_{x,i}$

Tel que $\Theta$ est le seuil minimal de similarité requise entre deux fonctionnalités pour qu'elles soient identiques. Nous avons fixé la valeur de $\Theta$ expérimentalement à $\Theta = 0.45$. Dans l'exemple précédent, on trouve que :

- $mostSimilarFeature(f_{1,1}, P_2) = f_{2,1}$
- $mostSimilarFeature(f_{2,1}, P_1) = f_{1,1}$
- $S(f_{1,1}, f_{2,1}) = 0.46 \geq \Theta$

Donc nous pouvons déduire que les deux fonctionnalités $f_{1,1}$ et $f_{2,1}$ faisant partie, respectivement, de $P_1$ et $P_2$ sont « *identique*s » et représentent deux implémentations différentes de la même fonctionnalité dans les deux variantes.

$\boldsymbol{computeIdenticalFeatures(f)}$ **:** Une fonction qui calcule pour une fonctionnalité $f$ un ensemble $A$ de fonctionnalités qui lui sont identiques dans les autres variantes de l'ensemble $V$. Formellement, $A = \{x_1, x_2, \dots, x_l\}$ tel que :

1. $\forall\ x_i\ \in A, \forall\ x_j\ \in A$, alors $x_i\ \in Cf_i$ et $x_j\ \in Cf_j$ et $i \neq j$
2. $|A| = |V| - 1$ ;
3. $\forall\ x \in A$, alors $isIdenticalFeature(f,\ x) = vrai.$

Dans l'exemple précédent, on trouve :

$$\mathrm{computeIdenticalFeatures}(\mathrm{f}_{1,1}) = (\mathrm{f}_{2,1}, \mathrm{f}_{3,1})$$

$\boldsymbol{isValidCommonality}(\boldsymbol{commonality})$ **:** Ayant idée sur ces différentes notions précédentes, nous arrivons donc à définir une commonalité. Une commonalité candidate pour une ensemble $V$ de variantes logicielles analysées est représentée sous forme d'un *n-uplet* de fonctionnalités distinctes, noté $commonality = (x_1, x_2, \dots, x_l)$. La fonction booléenne $isValidCommonality(commonality)$ vérifie

donc si une commonalité candidate, notée $commonality$, est une commonalité valide et retourne une valeur « $vrai$» ssi:

1. $\forall\, x_i \in commonality$, $\forall\, x_j \in commonality$, alors $x_i \in Cf_i$ et $x_j \in Cf_j$ et $i \neq j$
2. $|commonality| = |V|$ ;
3. $\forall\, x_i \in commonality$, $\forall\, x_j \in commonality$, et $i \neq j$, alors $identicalFeatures(x_i, x_j) = vrai$
4. Soit $COM_{fct}$ l'ensemble des commonalités construites pour un ensemble $V$ de variantes logicielles, $\forall\, com_i \in COM_{fct}$, $\forall\, com_j \in COM_{fct}$ et $i \neq j$ alors $com_i \cap com_j = \emptyset$.

Nous observons que la quatrième condition est assurée automatiquement par la troisième. Une fonctionnalité x composante d'une *commonalité valide* doit être exclusivement identique aux autres composantes de cette commonalité. Même dans le cas où cette fonctionnalité x figure dans un autre n-uplet candidat, elle est surement non identique à toutes ses composantes et le n-uplet sera jugé, par conséquent, comme non valide. Dans l'exemple précèdent, on trouve :

- $\mathrm{com} = (\mathrm{f}_{1,1}, \mathrm{f}_{2,1}, \mathrm{f}_{3,1})$ tel que *:* $\mathrm{f}_{1,1} \in \mathrm{Cf}_1$ de $\mathrm{P}_1$, $\mathrm{f}_{2,1} \in \mathrm{Cf}_2$ de $\mathrm{P}_2$ et $\mathrm{f}_{3,1} \in \mathrm{Cf}_3$ de $\mathrm{P}_3$ *;*
- $|com| = |V| = 3$ ;
- $identicalFeatures(\mathrm{f}_{1,1}, \mathrm{f}_{2,1}) = vrai$ et $identicalFeatures(\mathrm{f}_{1,1}, \mathrm{f}_{3,1}) = vrai$ et $identicalFeatures(\mathrm{f}_{2,1}, \mathrm{f}_{3,1}) = vrai$.

Nous pouvons donc déduire que com est une commonalité valide qui contient trois fonctionnalités, représentant chacune une implémentation différente de la même fonctionnalité. La liste complète des commonalités dans l'exemple précédent est :

$$COM_{fct} = \{ (f_{1,1}, f_{2,1}, f_{3,1}), \quad (f_{1,2}, f_{2,2}, f_{3,3}) \}$$

***isValidVariability*(*variability*)** *:* Une variabilité candidate pour un ensemble $V$ de variantes logicielles analysées est représentée sous forme d'un *n-uplet* de fonctionnalités distinctes, noté $variability = (x_1, x_2, \dots, x_l)$. La fonction booléenne $isValidVariability(variability)$ vérifie donc si une variabilité candidate, notée $variability$, est une variabilité valide et retourne une valeur « $vrai$» ssi:

1. $\forall\, x_i \in variability$, $\forall\, x_j \in variability$, alors $x_i \in Cf_i$ et $x_j \in Cf_j$ et $i \neq j$;
2. $0 < |variability| < |V|$ ;
3. $\forall\, x_i \in variability$, $\forall\, x_j \in variability$, et $i \neq j$, alors $identicalFeatures(x_i, x_j) = vrai$ ;
4. Soit $VAR_{fct}$ l'ensemble des variabilité construites pour un ensemble $V$ de variantes logicielles, $\forall\, var_i \in VAR_{fct}$, $\forall\, var_j \in COM_{fct}$ et $i \neq j$ alors $var_i \cap var_j = \emptyset$ ;
5. Soit $COM_{fct}$ l'ensemble des commonalités construites pour un ensemble $V$ de variantes logicielles, $\forall\, com \in COM_{fct}$, $\forall\, var \in VAR_{fct}$ alors $com \cap var = \emptyset$.

La quatrième et la cinquième condition sont automatiquement assurées par la troisième. La liste des variabilités obtenues pour l'exemple précédent est donc :

$$VAR_{fct} = \{(f_{1,3}, f_{2,3}), \quad (f_{1,4}, f_{3,2})\}$$

Expliquons maintenant comment fonctionne notre algorithme. Pour simplifier, nous l'avons divisé en deux parties. La première partie, illustrée dans Algorithme 1, construit la liste des commonalités qui sont partagées par toutes les variantes logicielles. La deuxième partie, illustrée dans Algorithme 2, construit itérativement la liste de variabilités de l'ensemble $V$ de variantes logicielles. Par souci de concision, nous omettons les détails concernant les structures de données sous-jacentes utilisées pour représenter les différents concepts définis précédemment.

**Entrées** :- $CF = \{Cf_1, Cf_2, \dots, Cf_l\}$ un ensemble de catalogues de fonctionnalités relatifs aux variantes logicielles $P_i \in V$ ;
- Θ un seuil minimal de similarité ;
- $S$ une matrice de similarité inter-fonctionnalité des variantes de l'ensemble $V$ ;

**Résultat** : - une liste $COM_{fct}$ de commonalités ;

$COM_{fct} := \{\}$

« Choisir un catalogue $Cf_i \in CF$ t.q. $\forall\, Cf_x \in CF, |Cf_i| \leq |Cf_x|$ » ;

**Pour chaque** fonctionnalité $f_{i,j} \in Cf_i$ **faire** :
$commonality = computeIdenticalFeatures(f_{i,j}) \cup \{f_{i,j}\}$ ;
**Si** $(isValidCommonality(commonality) = vrai)$ **alors** :
$COM_{fct} = COM_{fct} \cup \{commonality\}$;
**Fin**
**Fin**

« Retourner $COM_{fct}$ » ;

**Algorithme. 1. Calculer la liste des commonalités d'un ensemble de variantes.**

Selon la définition donnée ci-dessus, une commonalité est représentée par un vecteur de fonctionnalités distinctes, et dont chacune des composantes du vecteur appartient à un catalogue d'une variante distincte, et les fonctionnalités composantes du vecteur sont identiques l'une à l'autre. Afin d'optimiser le calcul des commonalités, nous sélectionnons une variante logicielle $v_{selected}$ à partir de l'ensemble $V$ de telle sorte que la taille de son catalogue de fonctionnalité $Cf_{v_{selected}}$ soit minimale par rapport aux catalogues des autres variantes dans $V$. En effet, l'idée de base est que le nombre maximal de commonalités pour l'ensemble $V$ est atteint lorsque toutes les fonctionnalités dans $Cf_{v_{selected}}$ sont partagées avec les autres variantes et ce nombre est égale la taille de $Cf_{v_{selected}}$ que nous avons sélectionné. Ensuite, pour chaque fonctionnalité $f$ du catalogue $Cf_{v_{selected}}$, nous construisons une liste contenant les fonctionnalités uniques qui lui sont identiques dans les autres variantes de $V - \{v_{selected}\}$, y compris $f$. Le résultat obtenu pour chaque $f$ est donc une commonalité candidate que nous devons vérifier et l'ajouter, si elle est jugée comme valide, à la liste finale des commonalités. Une commonalité candidate est donc considérée comme valide si aucune de ses

composantes ne figure dans d'autres commonalités qui ont été déjà construites. Chaque commonalité valide possède un nom unique qui l'identifie, i.e. $commonality\_i$ , qui est donné automatiquement par le système.

Le deuxième algorithme construit l'ensemble de variabilités. Une variabilité est représentée par un vecteur de fonctionnalités qui sont identiques l'une à l'autre, et qui figurent seulement dans quelques produits. Donc, une variabilité candidate est semblable à une commonalité, avec la seule différence que la taille d'une variabilité est forcément inférieure à la taille de l'ensemble $V$. Ainsi, nous procédons au calcul de variabilités en construisant une liste de tous les n-uplet possibles (tel que $|V| > n > 0$) qui sont composés de fonctionnalités uniques par rapport à leurs n-uplets relatifs, et identiques l'une à l'autre. La variabilité candidate est donc jugée comme valide si aucune de ses composantes ne figure dans les variabilités et les commonalités construites précédemment.

**Entrées** :- $CF = \{Cf_1, Cf, \dots, Cf_l\}$ un ensemble de catalogues de fonctionnalités relatifs aux variantes logicielles $P_i \in V$ ;
- Θ un seuil minimal de similarité ;
- $S$ une matrice de similarité inter-fonctionnalités des variantes de l'ensemble V
- une liste $COM_{fct}$ de commonalités ;

**Résultat** : - une liste $VAR_{fct}$ de variabilité ;

$VAR_{fct} := \{\}$

**Pour chaque** catalogue $Cf_i \in CF$ **faire** :
**Pour chaque** fonctionnalité $f_{i,j} \in Cf_i$ **faire** :
$variability = computeIdenticalFeatures(f_{i,j}) \cup \{f_{i,j}\}$ ;
**Si** $(isValidVariability(variability) = vrai)$ **alors** :
$VAR_{fct} = VAR_{fct} \cup \{variability\}$;
**Fin**
**Fin**
**Fin**

« Retourner $VAR_{fct}$ » ;

**Algorithme. 2. Calculer la liste des variabilités d'un ensemble de variantes.**

Chaque variabilité valide possède un nom unique qui l'identifie, i.e. $variability\_i$, qui est donné automatiquement par le système.

Les deux algorithmes que nous avons proposés consistent donc à un mécanisme de partitionnement, dont l'ensemble d'objets à partitionner est composé des fonctionnalités appartenant à toutes les variantes logicielles analysées. La spécificité de notre algorithme proposé par rapport à un d'autres algorithmes de partitionnement vient de la notion de « *fonctionnalités identiques* » qui permet de calculer des partitions avec une grande précision. En effet, nous ne cherchons pas seulement à construire des regroupements de fonctionnalités similaires, mais qui

sont plutôt identiques. De plus, la taille exacte des partitions qui représentent des commonalités et la taille maximale d'une variabilité sont aussi fixées automatiquement par le système dès le début du processus de partitionnement.

Après avoir calculé la liste complète de variabilités et de commonalités pour l'ensemble $V$ de variantes analysées, nous remplaçons le nom de chaque fonctionnalité $f_{i,j}$ dans son catalogue $Cf_i$ par le label de la variabilité ou commonalité de laquelle elle fait partie. Ainsi les catalogues de fonctionnalités qui subissent cette normalisation dans le même exemple précédent deviennent :

- $\overline{Cf_1} = \{com_1,\ com_2,\ var_1,\ var_2\}$
- $\overline{Cf_2} = \{com_1,\ com_2,\ var_1\}$
- $\overline{Cf_3} = \{com_1, var_2\ , com_2\}$

Tel que :

- $com_1 = (f_{1,1}, f_{2,1}, f_{3,1})$
- $com_2 = (f_{1,2}, f_{2,2}, f_{3,3})$
- $var_1 = (f_{1,3}, f_{2,3})$
- $var_2 = (f_{1,4}, f_{3,2})$

Ce faisons, nous avons donc arrivé à normalisé les catalogues de fonctionnalités de toutes les variantes logicielles, ce qui facilite la construction d'une LdP. Afin de construire un MF de la ligne de produit, nous devons d'abord construire une table qui contient la liste de toutes les variantes logicielles avec leurs nouveaux catalogues normalisés de fonctionnalités. Cette nouvelle *Table de Catalogue de Fonctionnalités* (TCF) est de taille $|TCF| = l \times m$ tel que $l$ est le nombre de toutes les variantes logicielles et $m$ et le nombre total de toutes les fonctionnalités dans la LdP (après la normalisation), i.e. $m = \left|\overline{Cf_1} \cup \overline{Cf_2} \cup \overline{Cf_3}\ \right|$.

La table *Tab 5.2* illustre la table TCF calculée pour l'exemple précédent. Pour chaque variante logicielle (i.e. ligne) la table TCF indique la participation d'une fonctionnalité donnée (i.e. colonne) dans la configuration de cette variante.

A ce stade de travail, la tab TCF constitue des données d'entrée pour un algorithme de construction d'un MF de la ligne de produits, tel que l'algorithme développé par Haslinger [Haslinger, E.N. et al. 2011, Haslinger, E.N. et al. 2013].

| | ***com₁*** | ***com₂*** | ***var₁*** | ***var₂*** |
|---|---|---|---|---|
| ***P₁*** | X | X | X | X |
| ***P₂*** | X | X | X | |
| ***P₃*** | X | X | | X |

**Tab 5.2 : Exemple de la table *TCF*.**

## 5.3 Evaluation de l'approche

Cette section présente les points forts qui caractérisent l'utilité de notre approche, ainsi que des points faibles qui limitent son utilisation dans certains contextes. Le tableau ci-dessous (*Tab. 5.3*) donne un bref résumé de notre approche en illustrant les phases couvertes du processus de réingénierie, les techniques utilisées, les types des entrées/résultats, … etc.

| | |
|---|---|
| **Phase de réingénierie** | ☑ Rétro-ingénierie |
| **Techniques utilisées** | ☑ Analyse statique<br>☑ Apprentissage automatique (Clustering)<br>☑ Indexation sémantique (Ontologie WordNet) |
| **Nombre de variantes logicielles** | ☑ Plusieurs variantes logicielles |
| **Hétérogénéité des variantes logicielles** | ☑ Hétérogénéité au niveau du vocabulaire utilisé<br>☑ Hétérogénéité au niveau du langages de programmation |
| **Granularité de la variabilité** | ☑ Classes<br>☑ Variables de classes<br>☑ Corps des méthodes |
| **Type de similarité** | ☑ Similarité structurelle (basée sur graphe de dépendance)<br>☑ Similarité sémantique (Basée sur WordNet) |
| **Données d'entrée** | ☑ Bytecode Java |
| **Données de sorties** | ☑ Liste de fonctionnalités pour chaque variante logicielle<br>☑ Liste des commonalités de la LdP<br>☑ Liste des variabilités de la LdP<br>☑ Traçabilité entre les fonctionnalités et les EdP qui les composent |
| **Chevauchement de fonctionnalités** | ☑ Prise en charge du chevauchement des fonctionnalités |
| **Contraintes complexes : inclure, exclure** | ☒ Pas de contraintes complexes entre les fonctionnalités |
| **Contraintes de sélection (OR, XOR, obligatoire)** | ☒ Pas de contraintes de sélection |

**Tab 5.3 : Caractéristiques de notre approche de rétro-ingénierie des LdP.**

### 5.3.1 Avantages

Notre approche d'extraction de variabilités et de commonalités d'une LdP est basée essentiellement sur notre technique d'extraction de fonctionnalités d'un système, proposée dans le chapitre 4. En plus des avantages qui sont donc apportés automatiquement par cette dernière (voir section 4.3.1), la force clé de notre nouvelle approche est la possibilité de défier le problème d'hétérogénéité dans les variantes logicielles au niveau du code en se basant sur la sémantique cachée derrière leur code. En défiant le problème d'hétérogénéité, notre approche est susceptible donc d'accélérer la migration d'une entreprise vers une solution LdP afin de profiter des avantages apportés par cette stratégie. Cette migration peut être effectuée en capitalisant sur les systèmes logiciels qui ont été déjà créés dans cette entreprise. Cela permettra donc de conserver le savoir-faire des experts qui ont participé au développement des anciens systèmes, tout en réduisant le taux d'investissement initial en termes de temps, de dépenses et d'efforts. Aussi, la portée de la nouvelle LdP peut être augmentée d'une façon incrémentale afin de répondre aux besoins du marché. Pour ce faire il suffit juste d'ajouter d'autres variantes logicielles à l'entrée du processus, sans avoir à s'inquiéter du problème d'hétérogénéité.

La sémantique cachée derrière le code des variantes logicielles analysées est identifiée utilisant l'ontologie WordNet. L'ontologie WordNet est considérée comme un standard et a été largement exploitée par les chercheurs dans le domaine de RI comme une couche sémantique. Etant donné la richesse des techniques et approches basées sur WordNet dans la littérature, notre approche peut encore tirer profit des avancées apportées par ces approches proposées dans cette tendance de recherche. Analogiquement, notre approche est donc simple à assimiler et à exploiter par la communauté WordNet.

Notre approche couvre actuellement deux étapes essentielles du processus de réingénierie : (1) la détection des fonctionnalités, et (2) l'analyse des fonctionnalités détectées pour construire l'ensemble de variabilités et de commonalités. Ces deux étapes consistent à la rétro-ingénierie d'une LdP. La navigation d'une fonctionnalité vers les EdP qui la composent représente un atout fort pour notre approche en favorisant la traçabilité. Bien que l'étape finale de réingénierie consistant à la transformation des commonalités et des variabilités en une LdP n'est pas encore couverte par notre travail actuel, la généricité et l'aspect formel des résultats obtenus facilitera une extension future de notre approche, ainsi que son intégration dans un processus existant de réingénierie des LdP. En effet, notre approche est complémentaire à d'autres techniques, et peut être utilisée pour construire des données d'entrée afin de construire un MF d'une LdP comme, par exemple, dans le cas de la technique proposée par Haslinger et al. [Haslinger, E.N. et al. 2011, Haslinger, E.N. et al. 2013].

### 5.3.2 Limites

L'approche proposée dans ce chapitre est basée principalement sur le calcul de similarité entre des groupes de concepts extraits à partir du code. Il est donc évident que les limitations majeures de notre approche sont dues en premier lieu aux limitations des techniques utilisées, i.e. (1) le calcul de similarité sémantique et (2) l'extraction des concepts utilisant le traitement automatique du langage naturel.

***La tokanization :***

Les outils d'analyse et de maintenance des logiciels permettant de réaliser des tâches telles que la traçabilité entre le code et la documentation, la localisation et l'extraction de fonctionnalités, et la réutilisation des programmes peuvent bénéficier des informations en langage naturel qui sont incorporées dans les identificateurs et commentaires. En effet, les noms d'EdP sont utilisés pour transmettre des concepts de domaine qui sont utiles et qui facilitent la compréhension et la maintenance du programme. Une première étape cruciale dans l'analyse des mots que les programmeurs utilisent est de diviser avec précision chaque nom d'EdP en ses mots composant. Dans notre approche, nous avons exploité des conventions de nommages, telles que les CamelCase et les underscores, afin de faciliter cette tâche. Néanmoins, ces conventions ne sont pas toujours respectées dans certaines situations et peuvent être modifiées par les programmeurs pour améliorer la lisibilité ce qui limite l'utilisation de l'approche proposée pour de tels cas.

***La désabréviation :***

Il est habituel, depuis des décennies, d'utiliser des noms courts et abrégés pour les EdP tels que les variables, les méthodes, …etc. Cette tradition est causée, au moins partiellement, par le fait que les premiers compilateurs, contrairement aux compilateurs modernes, limitaient la longueur d'un nom d'EdP à un certain petit nombre de caractères, comme l'affirme l'auteur dans [Laitinen, K. et al. 1997]. Ainsi, on trouve que les anciens programmes contiennent des noms plus abrégés que les programmes réalisés récemment.

Il existe des systèmes logiciels qui sont toujours utilisés depuis plusieurs décennies, et qui devraient être encore maintenus. Ces systèmes contiennent les connaissances implémentées par leurs développeurs, et peuvent être aussi considérés durant l'extraction d'une LdP, d'où provient l'intérêt de la «*désabriviation*» pour notre approche. Le terme « désabriviation » dénote le processus de remplacement des abréviations par des séquences de mots naturels [Laitinen, K. et al. 1997]. Selon Chikofsky [Chikofsky, E.J. and Cross, J.H. 1990] la désabriviation peut être vue comme un outil de redocumentation qui produit une meilleur version d'un programme source existant. Etant donné que les codes sources sont des textes techniques très uniques, leur désabréviation nécessite alors

un outil qui est spécifique pour un langage de programmation donné. Etant donné que le problème de désabriviation lui-même est un axe de recherche et qu'il n'est pas l'objectif de notre travail actuel, nous avons essayé donc d'étudier la possibilité d'intégrer un outil existant pour effectuer cette tâche.

On trouve peu d'approches qui ont été proposés jusqu'à maintenant pour la désabréviation. Une première méthode naïve consiste à créer un dictionnaire des abréviations communes. Cependant cette méthode est exposée au problème de *sur-spécialisation* (*over-fitting*), où les dictionnaires sont limités aux abréviations connues par leurs constructeurs et, par conséquent, ne sont pas génériques et réutilisables. De plus, selon le contexte dans lequel apparait une abréviation, cette dernière peut avoir plusieurs expansions possibles (candidats). Par exemple, « comp » peut signifier « compare » ou bien « component ». Certaines études ont essayé alors d'opter pour une approche hybride qui utilise des dictionnaires pré-constuits avec d'autres dictionnaires qui sont construit lors du processus de désabriviation.

Laitinen [Laitinen, K. et al. 1997] a développé un outil qui s'appelle *InName*[14]. Cet outil a été développé en Prolog et son exécution repose sur deux types de dictionnaires. Le premier dictionnaire contient quelques centaines de mots naturels qui sont prédéfinis par les créateurs de l'outil. Le second dictionnaire est construit utilisant les combinaisons de mots extraits à partir des noms d'EdP, à partir des commentaires, ou des mots qui sont suggérés par l'utilisateur durant le processus de désabréviation. L'outil InName est semi-automatique et dédié aux programmes écrits en C, et nécessite, par conséquent, un effort supplémentaire pour l'adapter aux programmes Java. InName est un outil interactif est intelligent vu qu'il est écrit en Prolog ce que lui permet d'apprendre et enrichir sa base de données à partir des feedbacks renvoyés par l'utilisateur. L'approche de Hill et al. [Hill, E. et al. 2008] utilise, en plus des noms d'EdP, l'information contextuelle dans les programmes Java tels que les commentaires et le javadoc, afin de trouver des expansion pour les abréviations. Cette approche semble être meilleur que celle de Laitinen [Laitinen, K. et al. 1997] et celle de Lawrie [Lawrie, D. et al. 2007] ; L'approche de Hill est totalement automatique et elle est implémentée sous forme d'un plugin[15] eclipse. L'approche de Lawrie [Lawrie, D. et al. 2007] retourne un résultat d'expansion s'il existe seulement une expansion possible, et ne traite pas, contrairement à l'approche de Hill [Hill, E. et al. 2008], le cas d'expansion multiple (plusieurs candidats pour la même abbréviation).

Malgré la précision des résultats de certaines approches, telle que celle de Hill [Hill, E. et al. 2008], et malgré la disponibilité de leurs outils [Laitinen, K. et al. 1997, Hill, E. et al. 2008], on ne peut pas les utiliser pour le moment dans notre approche afin

[14] http://www.naturalprogramming.com/inname/
[15] http://www.cis.udel.edu/~hill/amap

de défier le problème d'abréviation vu que ces outils s'exécutent sur le code source Java et utilise, en plus de noms d'EdP, les commentaires et le javadoc, quelque chose qu'on ne peut pas trouver dans le byte code utilisé dans notre approche.

### *La radicalisation :*

Dans notre technique d'extraction des LdP, nous avons utilisé l'outil Porter Stemmer pour la radicalisation de termes durant la phase d'extraction de concepts descriptifs, afin d'éviter des représentations différentes d'un concept en raison de flexions d'un mot, et réduire, par conséquent, l'espace de recherche. Porter Stemmer est classé par Wiese et al. [Wiese, A. et al. 2011] comme un outil *léger* qui favorise la sous-radicalisation (i.e. *understemming*) en ne produisant pas la même racine pour tous les mots relatifs au même concept. Par exemple les mots « add» et « adding » ne sont pas confondus au même mot. En outre, cet outil peut produire dans certain cas des mots incohérents qui ne font pas partie du langage naturel. Cet outil a été créé originalement pour être utilisé dans le domaine de RI, alors que les codes sources sont des textes techniques très uniques qui différent grandement des documents textuels. Les lacunes produites par l'utilisation de cet outil pour une tache d'analyse de programme peuvent donc causer des pertes d'information, ce qui heurte à la précision et la qualité des résultats de notre approche.

### *Le problème de termes interactifs*

La technique d'extraction de concepts, décrite dans la section 5.2.1, ne fait aucune distinction entre les concepts qui sont spécifiques au domaine du système et ceux de la mise en œuvre, tels que les concepts de l'interface graphique (GUI). Il est donc inévitable d'obtenir des termes tel que « *click »* et « *menu »* qui sont utilisés pour implémenter les éléments de GUI. Même que ces termes sont pertinents pour la description de la solution réalisée via le code, ils ne sont pas des termes de domaine. Ces termes interactifs peuvent être considérés comme des bruits et sont susceptible d'influencer les résultats.

## 5.4 Conclusion

Dans ce chapitre nous avons présenté notre nouvelle approche pour la configuration d'un ensemble de variantes logicielles, qui sont hétérogènes au niveau de code, en une ligne de produits logiciels. Nous avons utilisé un ensemble de techniques issues essentiellement du domaine d'apprentissage automatique et de la recherche d'information. Notre approche dérive, pour un ensemble de variantes logicielles, la liste exhaustive des variabilités et des commonalités qui caractérisent ces variantes, tout en gardant la traçabilité entre ces fonctionnalités et les EdP en relation.

# Chapitre 6: APPLICATION ET VALIDATION

Ce chapitre présente les différents mécanismes qui sont utilisés afin de valider nos approches proposées dans les chapitres 4 et 5, ainsi que les résultats obtenus. Ces mécanismes de validation ne sont évidemment pas les mêmes pour les deux techniques proposées vu les spécificités de chacune d'entre elles. En effet, même que les deux approches sont complémentaires, chacune d'entre elle adresse un problème différent dans le processus de rétro-ingénierie des LdP.

## 6.1 La découverte des fonctionnalités

Nous avons intégralement implémenté les étapes décrites dans le chapitre 4 comme un outil Java. Nous avons essayé de développer le moteur d'inférence de fonctionnalités comme étant un composant indépendant afin qu'il puisse être réutilisé d'une manière générique et efficace. Le moteur d'inférence de fonctionnalités parcourt les graphes de dépendances qui sont générés à l'aide d'un analyseur statique et cherche à découvrir les fonctionnalités du logiciel. Dans nos expérimentations, nous avons testé notre outil sur deux programmes Java, mais des logiciels développés utilisant d'autres langages orientés objet (C++, C#, etc…) peuvent également être analysés à l'aide de notre outil, si leurs graphes de dépendances sont fournis dans des fichiers XML respectant la DTD[16] utilisée par l'outil DependencyExtractor. Les expérimentations ont été faites sur un PC Windows 7 avec un processeur Intel i5-4200U et 8G de RAM.

### 6.1.1 Cas d'études

Afin de valider l'approche proposée, nous avons mené des expérimentations sur deux différentes applications open-source Java: *Mobile Media*[17] et *Drawing Shapes*[18]. L'avantage d'avoir utilisé ces deux cas d'études est qu'ils implémentent la variabilité à différents niveaux. En outre, les documentations et les MF correspondants sont disponibles, ce qui facilite la comparaison avec nos résultats. De plus, en utilisant ces deux cas d'études, nous ciblons deux catégories différentes de MF: (1) un MF plat qui correspond au cas d'étude *Drawing Shapes*, et (2) un MF imbriqué qui correspond au cas d'étude *Mobile Media*. Les deux figures *Fig 6.1* et *Fig 6.2* présentent les MF correspondants aux deux cas d'études, suivant la notation proposée par Ferber et al. [Ferber, S. et al. 2002]. Pour valider nos résultats utilisant ces cas d'étude, les véritables partitions (i.e. listing des EdP pour chacune des vraies fonctionnalités) ont été préparées manuellement en inspectant la documentation fournie par leurs auteurs.

[16] http://depfind.sourceforge.net/dtd/dependencies.dtd
[17] http://homepages.dcc.ufmg.br/~figueiredo/spl/icse08/
[18] https://code.google.com/p/svariants/

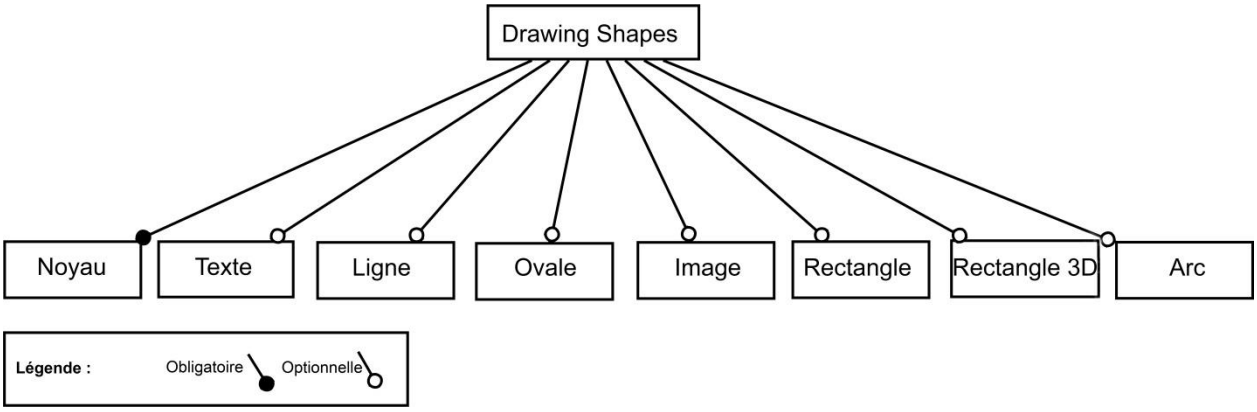



**Fig 6.1 : Le MF correspondant à Drawing Shapes.**

### 6.1.1.1 Drawing Shapes

La LdP *Drawing Shapes* représente un petit cas d'étude (la version 5 est composée de 8 package et 25 classes avec environ 0,6 KLOC). L'application *Drawing Shapes* permet à un utilisateur de dessiner sept types différents de formes dans une variété de couleurs. L'utilisateur choisit la forme et la couleur, puis il appuie sur le bouton de la souris et il fait glisser la souris pour créer la forme. L'utilisateur peut créer autant de formes qu'il désire. Les variantes logicielles de *Drawing Shapes* ont été développées utilisant la technique de copier-coller. Dans notre cas, nous utilisons la version 5 (la version complète) qui supporte les fonctionnalités *Rectangle-3D*, *Rectangle*, *Ovale*, *Texte*, *Arc*, *Image*, *Ligne*, et aussi *Noyau*.

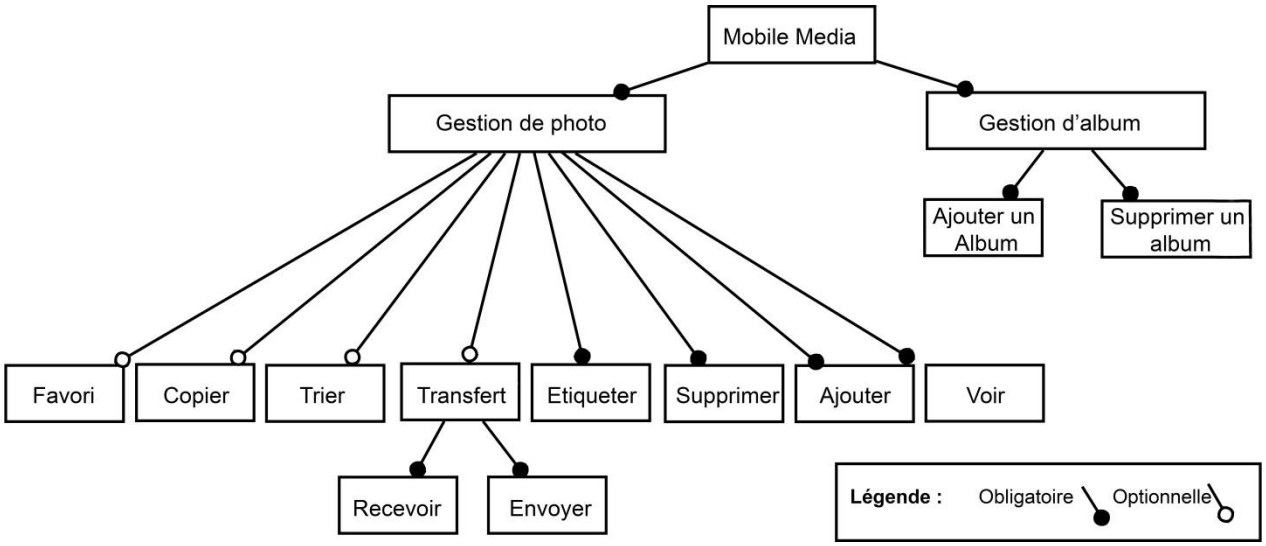



**Fig 6.2 : Le MF correspondant à Mobile Media**

### 6.1.1.2 Mobile Media

La LdP *Mobile Media* [Young, T.J. 2005] est un indicateur de référence (*benchmark*) qui est utilisé par les chercheurs dans le domaine de l'analyse de programmes et les recherches sur les LdP [Figueiredo, E. et al. 2008, Tizzei, L.P. et al. 2011, Al-

Msie'deen, R. et al. 2014]. Mobile Media permet la gestion des photos, de la musique et des vidéos sur des appareils mobiles, tels que les téléphones mobiles. Mobile Media a enduré sept scénarios d'évolution, qui ont conduit à huit versions, comprenant différents types de changements impliquant des fonctionnalités obligatoires, optionnelles, alternatives, ainsi que des caractéristiques non fonctionnelles. Dans cet exemple, nous avons utilisé la sixième version (R6) qui contient l'implémentation orientée objet de toutes les fonctionnalités optionnelles et obligatoires pour la gestion de photos. La version utilisée de Mobile Media est composée de 9 packages, 38 classes avec environ 3 KLOC.

Les EdP utilisés pour implémenter la fonctionnalité *Gestion d'exception* dans cette version de Mobile Media ont été écartés des données d'entrée dans la phase de prétraitement. En effet, comme nous l'avons expliqué précédemment, dans notre approche nous ne prenons en considération que les caractéristiques fonctionnelles. En outre, nous avons utilisé la valeur par défaut pour le seuil minimum de similarité d'OClustR, i.e. $\beta = 0$.

| **Cas d'étude** | **Fonctionnalité** | **Mesures d'évaluations** | | |
|---|---|---|---|---|
| | | **P %** | **R %** | **F %** |
| Drawing Shapes (version 5) | Noyau | 100 | 57 | 73 |
| | Texte | 100 | 100 | 100 |
| | Ovale | 100 | 100 | 100 |
| | Ligne | 100 | 100 | 100 |
| | Rectangle | 100 | 100 | 100 |
| | Image | 100 | 100 | 100 |
| | Arc | 100 | 100 | 100 |
| | Rectangle-3D | 11 | 33 | 17 |
| Mobile Media (version 6) | Ecran de démarrage | 100 | 100 | 100 |
| | Transfer SMS | 64 | 41 | 50 |
| | Gestion de photo | 100 | 35 | 52 |
| | Gestion d'album | 40 | 22 | 29 |
| | Voir photo | 54 | 58 | 56 |
| | Etiqueter photo | 70 | 39 | 50 |
| | Ajouter photo | 100 | 9 | 17 |
| | Supprimer photo | 40 | 40 | 40 |
| | Favori | 100 | 8 | 15 |
| | Trier | 100 | 8 | 15 |
| | Ajouter album | 67 | 22 | 33 |
| | Supprimer album | 38 | 38 | 38 |
| | Envoyer photo | 100 | 14 | 25 |
| | Recevoir photo | 40 | 17 | 24 |

**Tab 6.1 : Les fonctionnalités synthétisées depuis les applications Mobile Media et Drawing Shapes.**

### 6.1.2 Résultats et discussions

Notre outil implémenté a synthétisé les fonctionnalités rapidement, pour les deux cas d'études utilisés, dans un laps de temps raisonnable (environ 1 seconde). Le tableau *Tab 6.1* résume les résultats obtenus. Par souci de lisibilité, nous avons associé manuellement les noms des fonctionnalités aux partitions en fonction de leur contenu. Bien sûr, cela n'affecte pas la qualité de nos résultats.

Tout d'abord, nous observons que les valeurs F obtenus dans le cas d'étude Mobile Media ont été fortement influencées par les valeurs du rappel. Cependant, les valeurs de précision restent élevées pour la majorité des fonctionnalités, ce qui indique la pertinence des EdP qui les composent.

Ces observations peuvent être expliquées par le mécanisme de fonctionnement de OClustR, qui produit des partitions ayant une forte cohésion et un faible chevauchement. En effet, nous avons supposé que les fonctionnalités des applications analysées sont simulées en utilisant une ou plusieurs classes au niveau du code. Compte tenu de cette hypothèse, les implémentations de fonctionnalités de Mobile Media sont donc fortement chevauchées. En effet, de nombreuses classes sont partagées par les implémentations de plusieurs fonctionnalités simultanément, en raison de l'utilisation de plusieurs patrons de conception par les auteurs de Mobile Media, tels que *Modèle-Vue-Contrôleur* (MVC) et *chaîne de responsabilité*. Par exemple, les classes *PhotoController*, *ImageAccessor* et *ImageData* encapsulent toutes les méthodes de gestion de photos. Ainsi, des fonctionnalités telles que *Envoyer_Photo*, *Trier* et *Ajouter_Photo* partagent la plupart de leurs EdP (i.e. classes) et, par conséquent, ont atteint des faibles valeurs de rappel et de F. La fonctionnalité *Ecran_de_démarrage* quant à elle ne figure pas dans le MF original du cas d'étude *Mobile Media* mais notre outil a réussi quand même de l'extraire. Cette fonctionnalité, n'a pas souffert du problème causé par le chevauchement par ce que tous les traitements liés à cette fonctionnalité ont été encapsulés dans des classes distinctes. Ainsi, la fonctionnalité *Ecran_de_démarrage* a obtenu une valeur F maximale. Les conclusions tirées de l'analyse de Mobile Media sont conformes à celles obtenus pour le cas d'étude *Drawing Shapes*. Les fonctionnalités de *Drawing Shapes* sont légèrement chevauchées, de sorte que nous avons atteint une valeur F maximale, sauf pour les deux fonctionnalités *Rectangle-3D* et *Noyau*. La faible valeur de F pour *Rectangle-3D* est due à la faible cohésion entre ses classes au niveau du code. En effet, la vérification manuelle du code source du *Rectangle-3D* a révélé que l'auteur de *Drawing Shapes* a intentionnellement ou accidentellement causé cette faible cohésion en créant des classes pour *Rectangle-3D* sans pour autant créer des dépendances entre ces classes (i.e. les instanciations des classes sont manquantes). Ainsi, les EdP impliqués dans l'implémentation de *Rectangle-3D* ont été dispersés à travers les résultats, de sorte qu'une faible valeur de F a été atteinte. Cependant, dans le cas de la

fonctionnalité *Noyau*, certains EdP pertinents ont été extraits et reliés, par erreur, à une autre partition, de sorte que la valeur de F a été légèrement affectée. Quoi qu'il en soit, la valeur F de cette dernière est de 0,73 en moyenne, qui est une valeur acceptable.

| Variante logicielle | Précision BCubed | Rappel BCubed | FBCubed |
|---|---|---|---|
| Drawing Shapes | 80% | 67% | 73% |
| Mobile Media | 78% | 25% | 38% |

**Tab 6.2 : Résultats de l'évaluation BCubed des solutions de partitionnement avec chevauchement.**

L'évaluation de la solution de partitionnement avec chevauchement pour chacun des cas d'étude à l'aide de la mesure FBCubed confirme clairement les résultats obtenus (voir *Tab 6.2*). La valeur Précision BCubed pour le cas d'étude Mobile Media reste élevée, et la faible valeur du Rappel BCubed a affectée, par conséquent, la valeur globale de FBCubed. En revanche, on trouve que des valeurs équilibrées et élevées ont été atteintes pour les mesures BCubed de *Drawing Shapes*.

Hormis les valeurs de F et FBCubed qui s'avèrent appropriées, notre outil a généré un petit nombre de partitions dont la plupart sont pertinentes. Cette dernière caractéristique rend les résultats faciles à comprendre et efficacement manipulables par l'utilisateur. Le tableau *Tab 6.3* montre que les résultats pertinents générés par notre outil pour *Drawing Shapes* représentent 50% du total des résultats obtenus, et 88% pour *Mobile Media*.

Les résultats montrent que notre approche proposée pour l'extraction de fonctionnalités a généré un nombre raisonnable de fonctionnalités avec une précision acceptable. Selon les expérimentations que nous avons menées, notre méthode proposée fonctionne efficacement lorsqu'il s'agit de programmes ayant un MF plat.

| Variante logicielle | Clusters dérivés | Clusters pertinents | Taux de pertinence |
|---|---|---|---|
| Drawing Shapes | 16 | 8 | 50% |
| Mobile Media | 16 | 14 | 88% |

**Tab 6.3 : Nombre de correspondances pertinentes pour l'extraction de fonctionnalités.**

## 6.2 La rétro-ingénierie des LdP

Nous avons implémenté les étapes décrites dans le chapitre 5 comme un outil Java. Le moteur d'inférence de variabilités et de commonalités analyse le code de

fonctionnalités qui sont construites précédemment par notre approche présentée dans le chapitre 4. Le moteur d'inférence de variabilités calcule les similarités sémantiques en se basant sur la version 3.0 de l'ontologie WordNet. Pour ce faire, nous avons utilisé deux API Java : *Java WordNet Interface* (JWI) et *Java WordNet ::Similarity* (JWS), mais il est toujours possible de choisir parmi d'autres API similaires tel que *Semantic Measure Library*[19] (SML). Pour chacune des variantes logicielles, le moteur d'inférence de variabilités parcourt la liste des EdP relatifs à chaque fonctionnalité afin construire une liste des concepts (des sens) représentatifs. Ces listes de concepts sont ensuite utilisées pour mesurer la similarité entre les fonctionnalités, afin de pouvoir découvrir les variabilités. Les variantes logicielles utilisées dans le cadre de notre travail sont codées utilisant le langage Java. Dans le cas de variantes logicielles codées avec d'autres langages orientés objet, il sera toujours possible de les analyser si leurs graphes de dépendances sont fournis dans des fichiers XML respectant la DTD utilisée par l'outil *DependencyExtractor*. Les expérimentations ont été faites sur un PC Windows 7 avec un processeur Intel i5-4200U et 8G de RAM. Nous avons exécuté notre outil utilisant les paramètres suivants :

1. Le seuil minimal de similarité entre les fonctionnalités $\Theta$ fixé expérimentalement à $\Theta = 45$ ;
2. Le seuil minimal de représentativité d'un concept $minRep$ fixé expérimentalement à $minRep = 0.58$ ;
3. Le coefficient des concepts qui sont extraits à partir des noms de variables ou de méthodes fixé expérimentalement à $w_1 = 1$ ;
4. Le coefficient des concepts qui sont extraits à partir des noms de classes fixé expérimentalement à $w_2 = 20$ ;
5. Pour les paramètres $\alpha$ et $\beta$, qui sont utilisés dans le calcul de représentativité d'un concept, nous utilisons les valeur par défaut fixés par Desmontils à $\alpha$=1 et $\beta = 2$ [Desmontils, E. and Jacquin, C. 2002].

### 6.2.1 Cas d'étude

Afin de valider l'approche proposée, nous avons besoin d'exécuter notre outil implémenté sur un ensemble contenant au moins deux variantes logicielles qui adressent toutes les deux le même domaine d'application. Nous nous limitons dans le cadre de cette expérimentation, par souci de simplicité, à trois variantes seulement. La première variante utilisée est la version 5 de l'outil *Drawing Shapes* présenté précédemment (voir section 6.1.1.1). La version 5 de *Drawing Shapes* est la version complète qui contient huit fonctionnalités : *Noyau*, *Image*, *Ligne*, *Ovale*, *Texte*, *Rectangle*, *Rectangle-3D* et *Arc*. En ce qui concerne la deuxième et la troisième variante, nous les avons dérivés, respectivement, de la version 2 et la version 4 de l'outil *Drawing Shapes* en changeant manuellement les noms d'EdP utilisés dans le code source. La version 4 de *Drawing Shapes* contient quatre

[19] http://www.semantic-measures-library.org/sml/

fonctionnalités : *Noyau*, *Image*, *Ligne*, et *Rectangle*. La version 2 contient cinq fonctionnalités qui sont : *Noyau*, *Image*, *Ligne*, *Texte* et *Arc.* Le tableau *Tab 6.4* décrit la liste des fonctionnalités qui sont implémentées par chacune des variantes utilisées.

| | **Variante $P_1$** | **Variante $P_2$** | **Variante $P_3$** |
|---|---|---|---|
| | Drawing Shapes V2 | Drawing Shapes V4 | Drawing Shapes V5 |
| *Noyau* | X | X | X |
| *Ligne* | X | X | X |
| *Arc* | X | | X |
| *Rectangle* | | X | X |
| *Image* | X | X | X |
| *Texte* | X | | X |
| *Ovale* | | | X |
| *Rectangle-3D* | | | X |

**Tab 6.4 : Matrice d'implémentation de fonctionnalités par variante logicielle.**

| **Variante $P_1$** | **Variante $P_2$** | **Variante $P_3$** |
|---|---|---|
| `ArcedTrace` | | `Arc` |
| `MyIconForm` | `Photograph` | `MyImage` |
| `PaintingCustomizedForms` | `PaintingColoredGraphicalPatterns` | `DrawingShapes` |
| `ThinLinedTrace` | `StraightTracedLine` | `MyLine` |
| `ArcedTraceGradient` | | `ArcAngle` |
| `TextTypingZone` | | `MyText` |
| | `Parallelogram` | `MyRectangle` |
| `allowedFormsList` | `graphicalPatternsList` | `shapesArrayList` |
| `currentFormSort` | `actualPatternKind` | `currentType` |
| `FormCustomizationBoard` | `GraphicalPatternPaintingBoard` | `PaintJPanel` |
| `LinedTracePlacement` | `StraightTracedLinePlace` | `LinePosition` |
| `pathOfPrintedIcon` | `paintedPhotographPath` | `ImagePathToDraw` |

**Tab 6.5 : Exemple d'hétérogénéité introduite dans les noms d'EdP des variantes logicielles.**

Les trois variantes utilisées contiennent donc trois fonctionnalités communes (*Noyau*, *Image*, *Ligne*) et cinq fonctionnalités variables (*Ovale*, *Rectangle*, *Texte*, *Arc* et *Rectangle-3D*). En modifiant les noms d'EdP des versions 2 et 4, nous espérons donc introduire une certaine hétérogénéité au niveau du code utilisé dans les trois variantes logicielles. Cette hétérogénéité va donc servir à évaluer l'efficacité de notre système à comparer les fonctionnalités sur le plan sémantique. Le choix d'utiliser trois variantes de *Drawing Shapes* est justifié par le fait que la LdP *Drawing Shapes* est bien documentée par son auteur. De plus, le code de *Drawing Shapes* est plus simple, organisé et facile à comprendre et à modifier, et les fonctionnalités sont moins chevauchées au niveau du code que celles de *Mobile Media*.

Le tableau *Tab 6.5* illustre une liste non exhaustive des changements que nous avons introduits sur le code source des variantes logicielles. Chaque ligne du tableau montre les différentes façons de nommage du même EdP à travers les trois variantes logicielles analysées. Les cellules grisées dans le tableau indiquent que la fonctionnalité qui contient l'EdP en question n'est pas implémenté pas la variante logicielle relative.

| | **Variante $P_1$** | **Variante $P_2$** | **Variante $P_3$** |
|---|---|---|---|
| | Drawing Shapes V2 | Drawing Shapes V4 | Drawing Shapes V5 |
| *Noyau* | $f_{1,1}$ | $f_{2,1}$ | $f_{3,1}$ |
| *Ligne* | $f_{1,2}$ | $f_{2,2}$ | $f_{3,4}$ |
| *Arc* | $f_{1,4}$ | | $f_{3,7}$ |
| *Rectangle* | | $f_{2,3}$ | $f_{3,3}$ |
| *Image* | $f_{1,3}$ | $f_{2,4}$ | $f_{3,5}$ |
| *Texte* | $f_{1,5}$ | | $f_{3,2}$ |
| *Ovale* | | | $f_{3,6}$ |
| *Rectangle-3D* | | | $f_{3,8}$ |

**Tab 6.6 : Les fonctionnalités extraites à partir de l'ensemble des variantes logicielles.**

### 6.2.2 Résultats et discussions

Notre outil implémenté a synthétisé rapidement (environ 1 seconde) les fonctionnalités, ainsi que la liste de commonalités et de variabilité pour l'ensemble de variantes logicielles analysées. Le tableau *Tab 6.6* résume la liste des fonctionnalités extraites pour chaque variante, ainsi que le nom donné automatiquement par le système à chaque fonctionnalité. Les cellules grisées dans le tableau *Tab 6.6* indiquent que la fonctionnalité en question (i.e. ligne) n'est pas implémentée par la variante relative (i.e. colonne). Nous observons que notre outil

d'extraction de fonctionnalités a réussi à inférer toutes les fonctionnalités implémentées par chacune des variantes. Le calcul de la similarité sémantique entre toutes les fonctionnalités des variantes illustrées dans le tableau *Tab 6.6* a donné une matrice symétrique de similarité, nommée $S$, qui est présentée dans le tableau *Tab 6.7*. Comme nous l'avons mentionné précédemment, chaque fonctionnalité d'une variante logicielle est unique par rapport aux autres fonctionnalités appartenant à la même variante, et la similarité entre une fonctionnalité et elle-même est maximale (i.e. $S_{i,i} = 1$).

| | $f_{1,1}$ | $f_{1,2}$ | $f_{1,3}$ | $f_{1,4}$ | $f_{1,5}$ | $f_{2,1}$ | $f_{2,2}$ | $f_{2,3}$ | $f_{2,4}$ | $f_{3,1}$ | $f_{3,2}$ | $f_{3,3}$ | $f_{3,4}$ | $f_{3,5}$ | $f_{3,6}$ | $f_{3,7}$ | $f_{3,8}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f_{1,1}$ | 1 | | | | | 0.46 | 0.44 | 0.42 | 0.41 | 0.69 | 0.23 | 0.19 | 0.24 | 0.20 | 0.38 | 0.39 | 0.00 |
| $f_{1,2}$ | | 1 | | | | 0.24 | 0.91 | 0.59 | 0.42 | 0.28 | 0.40 | 0.21 | 0.64 | 0.22 | 0.32 | 0.39 | 0.00 |
| $f_{1,3}$ | | | 1 | | | 0.26 | 0.34 | 0.56 | 0.78 | 0.19 0 | 0.28 | 0.23 | 0.37 | 0.44 | 0.18 | 0.16 | 0.00 |
| $f_{1,4}$ | | | | 1 | | 0.20 | 0.31 | 0.20 | 0.19 | 0.36 | 0.08 | 0.07 | 0.08 | 0.06 | 0.28 | 0.49 | 0.00 |
| $f_{1,5}$ | | | | | 1 | 0.33 | 0.44 | 0.21 | 0.31 | 0.26 | 0.68 | 0.19 | 0.43 | 0.22 | 0.16 | 0.14 | 0.00 |
| $f_{2,1}$ | | | | | | 1 | | | | 0.57 | 0.42 | 0.14 | 0.25 | 0.24 | 0.11 | 0.10 | 0.00 |
| $f_{2,2}$ | | | | | | | 1 | | | 0.36 | 0.40 | 0.21 | 0.62 | 0.26 | 0.33 | 0.39 | 0.00 |
| $f_{2,3}$ | | | | | | | | 1 | | 0.28 | 0.18 | 0.25 | 0.24 | 0.21 | 0.45 | 0.38 | 0.00 |
| $f_{2,4}$ | | | | | | | | | 1 | 0.34 | 0.28 | 0.21 | 0.36 | 0.42 | 0.16 | 0.15 | 0.00 |
| $f_{3,1}$ | | | | | | | | | | 1 | | | | | | | |
| $f_{3,2}$ | | | | | | | | | | | 1 | | | | | | |
| $f_{3,3}$ | | | | | | | | | | | | 1 | | | | | |
| $f_{3,4}$ | | | | | | | | | | | | | 1 | | | | |
| $f_{3,5}$ | | | | | | | | | | | | | | 1 | | | |
| $f_{3,6}$ | | | | | | | | | | | | | | | 1 | | |
| $f_{3,7}$ | | | | | | | | | | | | | | | | 1 | |
| $f_{3,8}$ | | | | | | | | | | | | | | | | | 1 |

**Tab 6.7 : Matrice symétrique $S$ des similarités entre les fonctionnalités extraites à partir des variantes de Drawing Shapes**.

Après avoir analysé la matrice $S$ utilisant notre algorithme de calcul de commonalités (resp. variabilités) présenté dans le chapitre précédent (voir *Section 5.2.3*), nous avons obtenu les résultats illustrés dans le tableau *Tab 6.8*. Les résultats dans *Tab 6.8* montrent que nous avons réussi à dériver correctement deux

commonalités sur les trois commonalités prévues. La première commonalité « *Noyau* » est représentée par un triplé composé respectivement de la fonctionnalité *Noyau* de la variante $P_1$, la fonctionnalité *Noyau* de $P_2$ et la fonctionnalité *Noyau* de $P_3$. La deuxième commonalité « *Ligne*» est représenté respectivement par les fonctionnalités *Ligne* de $P_1$, *Ligne* de $P_2$ et *Ligne* de $P_3$.

| **Fonctionnalité** | **N-uplet dérivé** | **Evaluation** | | |
|---|---|---|---|---|
| | | **P %** | **R %** | **F %** |
| Les commonalités | | | | |
| Noyau | $(f_{1,1}, f_{2,1}, f_{3,1})$ | 100 | 100 | 100 |
| Ligne | $(f_{1,2}, f_{2,2}, f_{3,4})$ | 100 | 100 | 100 |
| Image | - | 00 | 00 | 00 |
| Les variabilités | | | | |
| Arc | $(f_{1,4}, f_{3,7})$ | 100 | 100 | 100 |
| Texte | $(f_{1,5}, f_{3,2})$ | 100 | 100 | 100 |
| Rectangle-3D | $(f_{3,8})$ | 100 | 100 | 100 |
| Rectangle | $(f_{3,3})$ | 100 | 50 | 66.67 |
| Oval | $(f_{2,3}, f_{3,6})$ | 50 | 50 | 50 |
| $F_{totale} \simeq 75.5\ \%$ | | | | |

**Tab 6.8 : Commonalités et variabilités inférées à partir des variantes de Drawing Shapes.**

Pour ce qui de la commonalité « *Image* », c'était prévu d'obtenir un triplé valide qui la représente, et qui est composé de $f_{1,3}$ de $P_1$, $f_{2,4}$ de $P_2$ et $f_{3,5}$ de $P_3$. Ce triplé candidat a été jugé par notre algorithme comme une commonalité non valide. En effet, comme nous l'avons expliqué dans la section 5.3.2, les composantes d'une commonalité candidate doivent être identiques l'une à l'autre pour que cette commonalité soit valide. Cependant, en observant les données présentées dans le tableau *Tab 6.7*, nous remarquons que :

- $S(f_{1,3},\ f_{2,4}) = 0.78$ et elles sont identiques ;
- $S(f_{3,5},\ f_{2,4}) = 0.42$ et $S(f_{3,5},\ f_{1,3}) = 0.44$. Ces deux valeurs sont très proches, mais reste quand même inférieurs, au seuil minimal utilisé : $\Theta = 45$ .

| Fonctionnalité | Termes dérivés | Concepts dérivés | Synset-ID | Représentativité |
|---|---|---|---|---|
| $f_{1,3}$ | path, set, form, get, icon, draw, placement | placement#1 | SID-01051331-N | 0.771 |
| | | placement#2 | SID-00039990-N | 0,617 |
| | | picture#1 | SID-03931044-N | 0,731 |
| | | path#1 | SID-00415676-N | 0,785 |
| $f_{2,4}$ | path, set, get, photograph, color, pattern, paint, graphic, placement | placement#1 | SID-01051331-N | 0,639 |
| | | path#1 | SID-00415676-N | 0,643 |
| | | picture#2 | SID-03925226-N | 0,647 |
| | | paint#1 | SID-03875218-N | 0,672 |
| $f_{3,5}$ | path, set, get, draw, e | path#1 | SID-00415676-N | 1,000 |
| | | path#2 | SID-03899328-N | 0,728 |
| | | path#3 | SID-08616311-N | 0,616 |
| | | path#4 | SID-09387222-N | 0,586 |
| | | draw#1 | SID-00115036-N | 0.606 |
| $f_{3,3}$ | set, e, get, draw | draw#1 | SID-00115036-N | 0,683 |
| $f_{2,3}$ | fit, set, color, parallelogram, pattern, paint, placement, graphic, return | parallelogram#1 | SID-13881175-N | 0,772 |
| | | placement#2 | SID-00039990-N | 0,673 |
| | | placement#1 | SID-01051331-N | 0,772 |
| | | pattern#1 | SID-00410247-N | 0,638 |
| | | fit#2 | SID-00555325-N | 0,660 |
| | | fit#1 | SID-14082788-N | 0,654 |

**Tab 6.9 : Exemple de termes et de concepts utilisés pour représenter les fonctionnalités dans les variantes logicielles.**

La composante $f_{3,5}$ n'est pas donc identique aux deux autres composantes du triplé candidat, ce qui rend ce dernier non valide, et il a été rejeté, par conséquent, de la liste finale des commonalités calculées. En essayons de détecter la cause derrière ce résultat, nous avons trouvé que les concepts utilisés pour indexer la fonctionnalité « *Image* » dans la troisième variante ne reflètent pas vraiment le rôle joué par cette fonctionnalité. Le tableau *Tab 6.9* illustre tous les termes qui ont été extrait à partir de la fonctionnalité « *Image* » dans les trois variantes analysées, ainsi que les concepts calculés par notre outil pour indexer les fonctionnnalités en question.

Les noms de certains EdP de la fonctionnalité $f_{1,3}$ contiennent le terme «*icon*», duquel le système a calculé le concept « *picture#1*» qui reflète bien le sens de cette fonctionnalité. De la même façon, les noms de certains EdP de la fonctionnalité $f_{2,4}$ contiennent les termes « *photograph*» et « *graphic* » à partir desquels le système a calculé le concept « *picture#2*» qui décrit bien la fonctionnalité. Les noms des EdP de la fonctionnalité $f_{3,5}$ contiennent le terme « *image*». Ce terme ne figure pas dans la liste des termes extraits à partir de la fonctionnalité en question par ce qu'il a été supprimé lors de l'extraction des termes représentatifs. En effet, durant l'étape de radicalisation, l'algorithme Porter Stemmer a calculé la racine « *imag*» pour le terme «image». Cette racine incohérente qui a été calculée ne fait pas partie du dictionnaire de WordNet et a été donc automatiquement rejetée durant la vérification de sa catégorie lexicale. Quoi qu'il en soit, nous avons réussi à extraire deux commonalités sur les trois prévues, avec une valeur F de 100%, ce qui est un résultat acceptable.

En plus de la liste des commonalités calculée pour les trois variantes logicielles analysées, notre système a aussi inféré la liste des variabilités (voir *Tab 6.8*). Les conclusions tirées des résultats concernant les commonalités sont conformes avec celles des variabilités. En effet, sur les cinq variabilités prévues, notre outil implémenté a réussi d'extraire correctement trois variabilités. Pour ce qui est de la variabilité *Rectangle*, c'était prévu d'obtenir un couple de fonctionnalités qui la représente et qui composé de $f_{2,3}$ de $P_2$ et $f_{3,3}$ de $P_3$. En analysant les données générées pour ces deux dernières fonctionnalités dans le tableau *Tab 6.9*, nous remarquons que la liste des termes extraits de la fonctionnalité $f_{3,3}$ ne contient aucun mot qui se réfère au concept « *rectangle*». En analysant le code de cette fonctionnalité, nous avons trouvé que les noms de certains de ses EdP contiennent le mot « *rectangle*». Néanmoins, l'algorithme Porter Stemmer a calculé pour ce mot une racine incohérente «*rectangl* » qui n'est pas reconnue par WordNet, et qui a été automatiquement rejetée, par conséquent, lors de la vérification de sa catégorie lexicale. Pour ce qui de la fonctionnalité $f_{2,3}$, la liste de ses terme contient le mot « *parallelogram* » à partir duquel le système a inféré le concept « *parallelogram#1* » qui décrit bien le rôle de cette fonctionnalité avec une valeur de représentativité de 77.2%. De toute façon, l'évaluation du couple de fonctionnalités calculé pour la variabilité « *Rectangle* » a donné une précision $P = 100\%$ et une valeur $F \simeq 67\%$ qui est toujours un résultat acceptable.

Pour ce qui est de la variabilité « *Ovale* », c'était prévu d'obtenir un 1-uplet qui la représente, et qui contient seulement la fonctoinnalité $f_{3,6}$ de la variante $P_3$. Cependant, nous avons obtenu un couple composé des fonctoinnalités $f_{3,6}$ de $P_3$ et de la fonctionnalité $f_{2,3}$ de $P_2$. La fonctionalité $f_{2,3}$ a été exclu du couple qui devait représenter la variabilité « *Rectangle* » comme nous venons de l'expliquer. Notre système a donc décidé que la fonctionnalité identique à $f_{3,6}$ est $f_{2,3}$ avec une valeur

de similarité $S(f_{3,6}, f_{2,3}) = 0.45$, une valeur qui est trop faible mais qui est quand même égale à la similarité minimale requise $\Theta$. La valeur F pour la variabilité « *Oval* » a était donc indirectement affectée par le choix de mauvais concepts représentatifs pour la fonctionnalité $f_{3,3}$, causé originalement par un mauvais calcul de la racine du terme « *Rectangle*» par Porter Stemmer. La valeur F pour la variabilité « *Oval* » est de 50%, un résultat qui reste toujours encourageant.

L'évaluation globale du système d'extraction des variabilités et des commonalités a donné une valeur $F_{totale} = 75.5\%$ qui est un résultat acceptable. Cette valeur de $F_{totale}$ a été donc influencée par les valeurs $F$ obtenues pour la commonalité « *Image* » et les deux variabilités « *Oval* » et « *Rectangle* », à cause du mauvais calcul des racines de certains termes par l'algorithme Porter Stemmer.

En plus de la valeur $F_{totale}$ de notre système qui est considérée comme appropriée, notre algorithme de classification a réussi de calculer un nombre réduit de de commonalités et de variabilités, qui sont presque toutes pertinentes. Le tableau *Tab 6.10* illustre le résumé de ces résultats. Sur l'ensemble de trois commonalités, notre outil a synthétisé deux commonalités qui sont toutes les deux correctes avec un taux de pertinence de 100%. De plus, les deux commonalités possèdent une valeur F de 100%, ce qui prouve encore sa performance. Notre outil a synthétisé sept variabilités dont cinq sont jugées comme pertinentes avec un taux de pertinence de 71.4%.

| | **Résultats calculés** | **Résultats pertinents** | **Taux de pertinence %** |
|---|---|---|---|
| **Commonalités** | 2 | 2 | 100 |
| **Variabilités** | 7 | 5 | 71.4 |

**Tab 6.10 : Nombre de correspondances pertinentes pour l'extraction de variabilités et de commonalités.**

## 6.3 Conclusion

Dans ce chapitre nous avons présenté les différentes étapes du processus expérimental que nous avons mené pour valider notre approche d'extraction de commonalités et de variabilités. Les résultats encourageants obtenus à partir de nos expérimentations ont montré l'utilité de notre première technique de découverte de fonctionnalité pour l'extraction d'un catalogue générique de fonctionnalités. De plus, selon les expérimentations menées sur notre deuxième approche d'extraction de commonalités et de variabilités à partir d'un ensemble de variantes logicielles,

notre outil implémenté s’est montré donc utile et performant en termes de temps d’exécution, de la précision des résultats et de leur pertinence, tout en défiant le problème d’hétérogénéité au niveau du code.

# Chapitre 7: CONCLUSION ET PERSPECTIVES

Dans ce chapitre, nous résumons nos principales contributions et nous discutons les orientations futures de notre recherche. Dans la section 7.1, nous présentons les principales contributions de notre l'approche. La section 7.2 présente les orientations futures et les opportunités pour étendre notre travail actuel.

## 7.1 Résumé des contributions

Les travaux présentés dans cette thèse s'articulent autour de l'extraction des LdP. Nous résumons nos contributions comme suit :

- Dans le chapitre 3, nous avons présenté un état de l'art sur les techniques d'extraction des LdP. Tout d'abord, nous avons présenté l'ensemble d'approches effectué dans le domaine de localisation de fonctionnalités. Ensuite nous avons analysé les approches d'extraction des LdP tout en faisant des comparaisons entre ces approches afin de cerner les points forts et les limitations de chacune d'entre elles.

- Dans le chapitre 4, nous avons proposé une nouvelle approche de localisation de fonctionnalités dans une variante logicielle qui consiste à l'extraction des implémentations de fonctionnalités à partir du code orienté objet. Nous avons utilisé les dépendances qui existent entre les éléments de programme au niveau du code afin d'appliquer un algorithme de clustering sur un graphe de dépendance d'une manière efficace. Dans notre approche, chaque cluster correspond à une fonctionnalité représentée par un ensemble d'éléments de programme. Nous avons testé notre outil implémenté pour extraire les fonctionnalités à partir du code de deux programmes Java. L'exécution de notre outil a montré que nous avons obtenu des résultats prometteurs qui sont compatibles avec les principaux objectifs de notre étude, ce qui rend l'approche proposée utile pour dériver la liste de fonctionnalités pour un logiciel existant.

- Dans le chapitre 5, nous avons proposé une nouvelle contribution qui consiste à l'extraction des commonalités et des variabilités d'une ligne de produits logiciels. La force clé de cette approche est qu'elle permet de configurer un ensemble de catalogues de fonctionnalités qui correspondent aux variantes logicielles en une ligne de produit. Les catalogues de fonctionnalités sont dérivés utilisant notre approche proposée dans le chapitre 4. Afin d'analyser les variantes logicielles qui sont hétérogènes au niveau du code, cette approche repose sur le traitement automatique du langage naturel ainsi que les mesures de similarité sémantique basées sur l'ontologie WordNet. Nous avons aussi proposé un algorithme de

partitionnement des fonctionnalités de l'ensemble de variantes logicielles en un ensemble de commonalités et de variabilités.

## 7.2 Perspectives de recherche

Nous avons plusieurs pistes et idées pour améliorer le travail réalisé dans cette thèse.

### *Améliorer le processus d'extraction de fonctionnalités*

Tout d'abord, nous aimerions améliorer notre approche d'extraction de fonctionnalités en utilisant d'autres techniques de classification avec chevauchement, afin de surmonter les limitations d'OClustR mentionnées précédemment. Pour ce faire, nous prévoyons de s'inspirer des travaux proposées pour la détection de communautés dans les réseaux sociaux. En effet, le principe de l'algorithme OClustR est de construire des clusters autours des meilleurs centres, ce qui a causé le phénomène des centres prédateurs. Ça sera donc mieux d'opter pour une approche dans laquelle on tend de trouver des meilleures communautés, chacune d'entre elles représente une fonctionnalité.

De plus, afin de défier le problème de complexité lors du calcul des mesures de similarité basées sur la distance structurelle entre les EdP, nous prévoyons d'utiliser des méthodes d'approximation basée sur les marches aléatoires (*random walks*)[Aggarwal, C.C. 2015], tels que la marche aléatoire avec redémarrage et l'algorithme SimRank [Jeh, G. and Widom, J. 2002]. En plus de la réduction de complexité, la mesure basée sur les marches aléatoires fournit un résultat qui est différent de celui de la mesure du plus court chemin par ce que la multiplicité des chemins entre une paire de nœuds est également considérée lors du calcul de la similarité. Une telle technique est susceptible d'améliorer la précision de nos résultats et de réduire l'effet du phénomène des ws-graphe (i.e. clusters) prédateurs.

Nous prévoyons également de documenter les fonctionnalités extraites, en déduisant automatiquement leurs noms, afin de faciliter leur interprétation et leur manipulation dans des tâches ultérieures. Pour ce faire, les vecteurs de concepts descriptifs qui ont été dérivés lors de calcul de similarités constituent un appui fort, et peuvent être utilisés comme des données d'entrée pour une technique de traitement de langage naturel, telle que le résumé automatique de texte (*text summarization*) ou l'extraction de sujet (*topic tracking*).

### *Améliorer la mesure de similarité sémantique*

Nous avons utilisé dans notre approche (i.e. chapitre 5) des mesures de similarité entre les paires de concepts, dont les résultats obtenus ont été utilisés pour quantifier la similarité entre les fonctionnalités extraites à partir de plusieurs variantes logicielles. Les données générées à cette étape de notre approche influent

donc grandement la qualité des résultats finaux. Ainsi, l'amélioration de la précision de cette technique est donc susceptible de raffiner les résultats et d'augmenter leurs pertinences. Pour ce faire, nous prévoyons d'utiliser d'autres mesures de similarité à base de WordNet telle que celle proposée par Seco et al. [Seco, N. et al. 2004].

### *Améliorer la précision des concepts extraits*

Afin d'éviter des représentations différentes d'un concept en raison de flexions des mots, et pour obtenir des concepts pertinents et précis durant la phase d'extraction de concepts, nous prévoyons d'étudier l'efficacité d'un outil *stricte* de radicalisation (i.e. *heavy stemmer*) qui est dédié au domaine d'analyse des logiciels, tel que l'outil MStem présenté dans [Wiese, A. et al. 2011]. Ceci est susceptible de surmonter le problème des racines incohérentes calculées par Porter Stemmer pour certains termes.

De plus, afin de corriger le problème des termes interactifs, mentionné précédemment, nous prévoyons d'introduire une intervention manuelle subtile et efficace, appelée la *sélection des mots clés du domaine*, telle que celle décrite dans [Abebe, S.L. and Tonella, P. 2015]. Ce processus de sélection de termes du domaine nécessite l'implication d'un développeur pour considérer (ou pas) un terme en tant qu'un véritable mot clé du domaine. Si le terme est jugé comme un véritable mot de domaine, il est maintenu dans l'index des termes qui sont utilisés pour extraire des concepts à l'étape suivante tandis que les autres sont enlevés. Cette sélection manuelle sera effectuée pour un sous-ensemble de termes qui sont sélectionnés automatiquement en utilisant une technique appropriée. Cela nécessite simplement de cocher/décocher chaque mot-clé candidat et nécessite, par conséquent, un minimum d'effort de la part de l'analyste.

En outre, nous prévoyons d'extraire des concepts ayant des labels plus longs (i.e. composés de plusieurs termes). En effet, des concepts de ce genre sont plus spécifiques et porteurs d'une quantité importante d'information et exprime bien, par conséquent, l'information incorporée dans le code [Baziz, M. et al. 2005, Vallet, D. et al. 2005]. Par exemple, le concept désigné par « polar bear » (i.e. ours polaire) est plus précis et informatif que celui qui est désigné seulement par le terme « bear ».

### *Les ontologies comme un formalisme de modélisation*

Etant donné que la modélisation de variabilité est au cœur de l'ingénierie des LdP, l'amélioration du formalisme de modélisation utilisé aura donc un effet considérable sur le processus de gestion de variabilité en générale. Les MF et les ontologies jouent le même rôle en offrant des méta-informations pour concevoir les modèles tout au long du développement des logiciels. De plus, les ontologies semblent être meilleures que les diagrammes de fonctionnalités en termes de

capacités de description ; un diagramme de fonctionnalités représente un sous-ensemble des notations d'une ontologie et peut, par conséquent, être considéré comme une vue sur une ontologie [Czarnecki, K. et al. 2006]. En effet, quelques fonctionnalités comme, par exemple, les attributs de référence et les associations sont en dehors de la portée des diagrammes de fonctionnalités et peuvent être mieux représentées avec les ontologies. D'autre part, étant basé sur XML, le format OWL facilite le stockage et l'échange des modèles de fonctionnalités. Un autre avantage est la disponibilité des techniques de raisonnement qui peuvent être utilisées pour vérifier automatiquement la consistance d'une configuration.

### *Utiliser l'analyse dynamique*

Etant donné que les caractéristiques fonctionnelles d'un logiciel sont associées à son comportement, nous avons l'intention d'enrichir les données d'entrée en utilisant des informations dynamiques. En effet, le comportement de certains types de systèmes ne peut être capturé qu'au moment de leurs exécutions. Notre approche actuelle est basée essentiellement sur l'analyse statique du code pour extraire des fonctionnalités. Même si elles sont fondées sur des stratégies différentes de fonctionnements, l'analyse dynamique et l'analyse statique peuvent être complémentaires dans certains points. Une autre étape d'analyse dynamique est donc indispensable pour compléter l'information statique afin d'inférer les associations qui existent entre les fonctionnalités et qui ne peuvent être détectées qu'en utilisant une analyse dynamique.  Ces associations vont servir de base pour construire un MF cohérent ainsi que pour l'induction des contraintes complexe.

# RÉFÉRENCES


[Abebe, S.L. et al. 2013] Abebe, S.L., Alicante, A., Corazza, A. and Tonella, P. Supporting concept location through identifier parsing and ontology extraction. *Journal of Systems and Software*, 86, 11 (2013), 2919-2938.

[Abebe, S.L. and Tonella, P. 2010] Abebe, S.L. and Tonella, P. Natural Language Parsing of Program Element Names for Concept Extraction. In *IEEE 18th International Conference on Program Comprehension (ICPC)* (2010), 156-159.

[Abebe, S.L. and Tonella, P. 2011] Abebe, S.L. and Tonella, P. Towards the Extraction of Domain Concepts from the Identifiers. In *18th Working Conference on Reverse Engineering (WCRE)* (2011), 77-86.

[Abebe, S.L. and Tonella, P. 2015] Abebe, S.L. and Tonella, P. Extraction of domain concepts from the source code. *Science of Computer Programming*, 98, P4 (2015), 680-706.

[Aggarwal, C.C. 2015] Aggarwal, C.C. Similarity and Distances. In: Data Mining: The text book, Springer International Publishing (2015), 63-91.

[Al-Msie'deen, R. et al. 2014] Al-Msie'deen, R., Huchard, M., Seriai, A.-D., Urtado, C. and Vauttier, S. Automatic Documentation of [Mined] Feature Implementations from Source Code Elements and Use-Case Diagrams with the REVPLINE Approach. *International Journal of Software Engineering and Knowledge Engineering*, 24, 10 (2014), 1413-1438.

[Al-Msie'Deen, R. et al. 2013] Al-Msie'Deen, R., Seriai, A.D., Huchard, M., Urtado, C. and Vauttier, S. Mining Features from the Object-Oriented Source Code of Software Variants by Combining Lexical and Structural Similarity. In *IEEE 14th International Conference on Information Reuse & Integration*, Las Vegas, NV, USA (2013), 586-593.

[Alpaydin, E. 2010] Alpaydin, E. Introduction to Machine Learning, The MIT Press (2010).

[Alves, V. et al. 2010] Alves, V., Niu, N., Alves, C. and Valença, G. Requirements engineering for software product lines: A systematic literature review. *Inf. Softw. Technol.*, 52, 8 (2010), 806-820.

[Amigó, E. et al. 2009] Amigó, E., Gonzalo, J., Artiles, J. and Verdejo, F. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Information Retrieval*, 12, 4 (2009), 461-486.

[Anitha Elavarasi, S. et al. 2014] Anitha Elavarasi, S., Akilandeswari, J. and Menaga, K. A survey on semantic similarity measure. *International journal of research in advent technology*, 2, 3 (2014).

[Araar, I.E. and Seridi, H. 2016] Araar, I.E. and Seridi, H. Software Features Extraction From Object-Oriented Source Code Using an Overlapping Clustering Approach. *Informatica*, 40, 2 (2016), 11.

[Ball, T. 1999] Ball, T. The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99* (1999), 216-234.

[Baziz, M. et al. 2005] Baziz, M., Boughanem, M. and Aussenac-Gilles, N. Conceptual Indexing Based on Document Content Representation. In *Proceedings of the 5th International Conference on Conceptions of Library and Information Sciences CoLIS 2005*, Glasgow, UK (2005), 171-186.

[Berard, E.V. 1993] Berard, E.V. Essays on object-oriented software engineering, Prentice Hall (1993).

[Biggerstaff, T.J. 1989] Biggerstaff, T.J. Design recovery for maintenance and reuse. *Computer*, 22, 7 (1989), 36-49.

[Bosch, J. 2000] Bosch, J. Design and use of software architectures: adopting and evolving a product-line approach, ACM Press/Addison-Wesley Publishing Co. (2000).

[Bosch, J. and Capilla, R. 2013] Bosch, J. and Capilla, R. Variability Implementation. In: Systems and Software Variability Management: Concepts, Tools and Experiences, Berlin, Heidelberg, Springer Berlin Heidelberg (2013), 75-86.

[Bosch, J. et al. 2001] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. and Pohl, K. Variability Issues in Software Product Lines. In *4th International Workshop on Software Product-Family Engineering PFE 2001*, Bilbao, Spain (2001), 13-21.

[Boubekeur, F. et al. 2010] Boubekeur, F., Boughanem, M., Tamine, L. and Daoud, M. Using WordNet for Concept-Based Document Indexing in Information Retrieval. In *The Fourth International Conference on Advances in Semantic Processing (SEMAPRO 2010)*, Florence, Italy (2010), 151-157.

[Chen, L. et al. 2009] Chen, L., Babar, M.A. and Ali, N. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, San Francisco, California (2009), 81-90.

[Chen, Y.-L. and Hu, H.-L. 2006] Chen, Y.-L. and Hu, H.-L. An overlapping cluster algorithm to provide non-exhaustive clustering. *European Journal of Operational Research*, 173, 3 (2006), 762-780.

[Chikofsky, E.J. and Cross, J.H. 1990] Chikofsky, E.J. and Cross, J.H. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7, 1 (1990), 13-17.

[Clements, P. and Northrop, L. 2001] Clements, P. and Northrop, L. Software product lines: practices and patterns, Addison-Wesley (2001).

[Czarnecki, K. et al. 2006] Czarnecki, K., Chang, H., Kim, P. and Kalleberg, K.T. Feature models are views on ontologies. In *10th International Software Product Line Conference* (2006), 41-51.

[Davis, S.M. 1987] Davis, S.M. Future perfect, Addison-Wesley (1987).

[Deelstra, S. et al. 2005] Deelstra, S., Sinnema, M. and Bosch, J. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74, 2 (2005), 173-194.

[Deissenbock, F. and Pizka, M. 2005] Deissenbock, F. and Pizka, M. Concise and consistent naming. In *13th International Workshop on Program Comprehension (IWPC'05)* (2005), 97-106.

[Desmontils, E. and Jacquin, C. 2002] Desmontils, E. and Jacquin, C. Indexing a web site with a terminology oriented ontology. The first International Semantic web working symposium (SWWS 2001), Stanford University, California, USA, IOS Press, 75 (2002), 181-197.

[Dietrich, J. et al. 2008] Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G. and Duchrow, M. Cluster analysis of Java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, Ammersee, Germany (2008), 91-94.

[Draszawka, K. and Szymański, J. 2011] Draszawka, K. and Szymański, J. External Validation Measures for Nested Clustering of Text Documents. In: Emerging Intelligent Technologies in Industry, Springer Berlin Heidelberg, 369 (2011), 207-225.

[Ernst, M.D. 2000] Ernst, M.D. Dynamically discovering likely program invariants, PhD thesis, University of Washington (2000).

[Ernst, M.D. 2003] Ernst, M.D. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis* (2003), 24-27.

[Eyal-Salman, H. et al. 2013] Eyal-Salman, H., Seriai, A.D. and Dony, C. Feature-to-Code Traceability in Legacy Software Variants. In *39th EUROMICRO Conference on Software Engineering and Advanced Applications* Santander, Spain (2013), 57-61.

[FAMILIES project website 2005] FAMILIES project website. (2005). "FAMILIES project." Retrieved 27 October, (2012), from http://www.esi.es/Projects/Families/index.html.

[Ferber, S. et al. 2002] Ferber, S., Haag, J. and Savolainen, J. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Proceedings of the Second International Conference on Software Product Lines* (2002), 235-256.

[Figueiredo, E. et al. 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F. and Dantas, F. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *30th International Conference on Software Engineering*, Leipzig, Germany (2008), 261-270.

[Finlayson, M.A. 2014] Finlayson, M.A. Java libraries for accessing the princeton wordnet: Comparison and evaluation. In *Proceedings of the 7th Global Wordnet Conference, Tartu, Estonia* (2014).

[Fisher, D. 1987] Fisher, D. Knowledge Acquisition Via Incremental Conceptual Clustering. *Machine Learning*, 2, 2 (1987), 139-172.

[Floyd, R.W. 1962] Floyd, R.W. Algorithm 97: Shortest path. *Communications of the ACM*, 5, 6 (1962), 345.

[Graaf, B. et al. 2006] Graaf, B., Weber, S. and van Deursen, A. Migrating supervisory control architectures using model transformations. In *The 10th European Conference on Software Maintenance and Reengineering* (2006), 153-164.

[Gurp, J.V. et al. 2001] Gurp, J.V., Bosch, J. and Svahnberg, M. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, Amsterdam, The Netherlands (2001), 45-54.

[Halmans, G. and Pohl, K. 2003] Halmans, G. and Pohl, K. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2, 1 (2003), 15-36.

[Harispe, S. et al. 2013] Harispe, S., Ranwez, S., Janaqi, S. and Montmain, J. Semantic Measures for the Comparison of Units of Language, Concepts or Instances from Text and Knowledge Representation Analysis. *A Comprehensive Survey and a Technical Introduction to Knowledge-based Measures Using Semantic Graph Analysis, LGI2P/EMA Research Center, Parc scientifique, France* (2013).

[Haslinger, E.N. et al. 2011] Haslinger, E.N., Lopez-Herrejon, R.E. and Egyed, A. Reverse Engineering Feature Models from Programs' Feature Sets. In *18th Working Conference on Reverse Engineering*, Limerick, Ireland (2011), 308-312.

[Haslinger, E.N. et al. 2013] Haslinger, E.N., Lopez-Herrejon, R.E. and Egyed, A. On extracting feature models from sets of valid feature combinations. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering*, Rome, Italy (2013), 53-67.

[Hernandez, N. 2005] Hernandez, N. Ontologies de domaine pour la modélisation du contexte en recherche d'information, PhD thesis, Université Paul Sabatier-Toulouse III (2005).

[Hill, E. et al. 2014] Hill, E., Binkley, D., Lawrie, D., Pollock, L. and Vijay-Shanker, K. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19, 6 (2014), 1754-1780.

[Hill, E. et al. 2008] Hill, E., Fry, Z.P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L. and Vijay-Shanker, K. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 international working conference on Mining software repositories*, Leipzig, Germany (2008), 79-88.

[Jeh, G. and Widom, J. 2002] Jeh, G. and Widom, J. SimRank: a measure of structural-context similarity. In *The 8th ACM international conference on Knowledge discovery and data mining* (2002), 538-543.

[Jiang, J.J. and Conrath, D.W. 1997] Jiang, J.J. and Conrath, D.W. Semantic similarity based on corpus statistics and lexical taxonomy. In *10th International Conference on Research in Computational Linguistics, ROCLING'97* (1997).

[Johansen, M.F. et al. 2010] Johansen, M.F., Fleurey, F., Acher, M., Collet, P. and Lahire, P. Exploring the Synergies Between Feature Models and Ontologies. In *International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010)*, Jeju Island, South Korea (2010), 163-170.

[Kang, K. et al. 1990] Kang, K., Cohen, S., Hess, J., Novak, W.E. and Peteron, A. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, United States (1990).

[Khan, L. and Luo, F. 2002] Khan, L. and Luo, F. Ontology construction for information selection. In *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2002)*, Washington, DC, USA (2002), 122-127.

[Khuller, S. and Raghavachari, B. 2009] Khuller, S. and Raghavachari, B. Basic graph algorithms. In: Algorithms and Theory of Computation Handbook, Second Edition, Volume 1, Chapman & Hall/CRC (2009), 7.1-7.24.

[Kiczales, G. et al. 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming ECOOP' 97*, Jyväskylä, Finland (1997), 220-242.

[Kowalski, G.J. and Maybury, M.T. 2002] Kowalski, G.J. and Maybury, M.T. Information Storage and Retrieval Systems: Theory and Implementation, Springer US (2002).

[Krueger, C.W. 2002] Krueger, C.W. Easing the Transition to Software Mass Customization. In: Software Product-Family Engineering : Revised Papers from the 4th International Workshop on Software Product-Family Engineering, Springer Berlin / Heidelberg, 2290 (2002), 282-293.

[Krueger, C.W. 2004] Krueger, C.W. Towards a taxonomy for software product lines. In: Software Product-Family Engineering, 3014 (2004), 323-331.

[Krueger, C.W. 2011] Krueger, C.W. (2011, 28 Mars 2011). "Introduction to Software Product Lines." Retrieved 09 October, (2012), from http://www.softwareproductlines.com/introduction/introduction.html.

[Laguna, M.A. and Crespo, Y. 2013] Laguna, M.A. and Crespo, Y. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming*, 78, 8 (2013).

[Laitinen, K. et al. 1997] Laitinen, K., Taramaa, J., Heikkilä, M. and Rowe, N.C. Enhancing maintainability of source programs through disabbreviation. *Journal of Systems and Software*, 37, 2 (1997), 117-128.

[Lawrie, D. et al. 2007] Lawrie, D., Feild, H. and Binkley, D. Extracting Meaning from Abbreviated Identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)* (2007), 213-222.

[Lin, D. 1998] Lin, D. An Information-Theoretic Definition of Similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning* (1998), 296-304.

[Lozano, A. 2011] Lozano, A. An Overview of Techniques for Detecting Software Variability Concepts in Source Code. In *ER 2011 Workshops*, Brussels, Belgium (2011), 141-150.

[Mabotuwana, T. et al. 2013] Mabotuwana, T., Lee, M.C. and Cohen-Solal, E.V. An ontology-based similarity measure for biomedical data - Application to radiology reports. *Journal of Biomedical Informatics*, 46, 5 (2013), 857-868.

[Manning, C.D. et al. 2008] Manning, C.D., Raghavan, P. and Schütze, H. Introduction to information retrieval, Cambridge university press, Cambridge, England (2008).

[Mihalcea, R. and Moldovan, D. 2000] Mihalcea, R. and Moldovan, D. Semantic indexing using WordNet senses. In *Proceedings of the ACL-2000 workshop on Recent advances in natural language processing and information retrieval: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 11* (2000), 35-45.

[Miller, G.A. 1995] Miller, G.A. WordNet: a lexical database for English. *Communications of the ACM*, 38, 11 (1995), 39-41.

[Miller, G.A. and Charles, W.G. 1991] Miller, G.A. and Charles, W.G. Contextual correlates of semantic similarity. *Language and cognitive processes*, 6, 1 (1991), 1-28.

[Mock, M. 2003] Mock, M. Dynamic analysis from the bottom up. In *Workshop on Dynamic Analysis (WODA 2003)*, Portland, Oregon, USA (2003), 13-16.

[Mohammad, S.M. and Hirst, G. 2012] Mohammad, S.M. and Hirst, G. Distributional measures of semantic distance: A survey. (A revised version of "Distributional Measures as Proxies for Semantic Relatedness"), University of Toronto (2012).

[Neighbors, J. 1986] Neighbors, J. The Draco approach to constructing software from reusable components. In: Readings in artificial intelligence and software engineering, Morgan Kaufmann Publishers Inc. (1986), 525-535.

[Niu, N. and Easterbrook, S. 2008] Niu, N. and Easterbrook, S. On-Demand Cluster Analysis for Product Line Functional Requirements. In *12th International Software Product Line Conference SPLC '08.* (2008), 87-96.

[Paralic, J. and Kostial, I. 2003] Paralic, J. and Kostial, I. Ontology-based information retrieval. In *Proceedings of the 14th International Conference on Information and Intelligent systems (IIS 2003), Varazdin, Croatia* (2003), 23-28.

[Parnas, D.L. 1976] Parnas, D.L. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2, 1 (1976), 1-9.

[Paskevicius, P. et al. 2012] Paskevicius, P., Damasevicius, R., Karciauskas, E. and Marcinkevicius, R. Automatic Extraction of Features and Generation of Feature Models from Java Programs. *Information Technology and Control*, 41, 4 (2012), 376-384.

[Pérez-Suárez, A. et al. 2013] Pérez-Suárez, A., Martínez-Trinidad, J.F., Carrasco-Ochoa, J.A. and Medina-Pagola, J.E. OClustR: A new graph-based algorithm for overlapping clustering. *Neurocomputing*, 121 (2013), 234-247.

[Perkins, J.H. and Ernst, M.D. 2004] Perkins, J.H. and Ernst, M.D. Efficient incremental algorithms for dynamic detection of likely invariants. In *ACM SIGSOFT Software Engineering Notes* (2004), 23-32.

[Pesquita, C. et al. 2007] Pesquita, C., Faria, D., Bastos, H., Falcão, A. and Couto, F. Evaluating gobased semantic similarity measures. *Proceedings of the 10th Annual Bio-Ontologies Meeting*, 2007 (2007), 37-40.

[Pesquita, C. et al. 2008] Pesquita, C., Faria, D., Bastos, H., Ferreira, A.E., Falcão, A.O. and Couto, F.M. Metrics for GO based protein semantic similarity: a systematic evaluation. *BMC bioinformatics*, 9, S-5 (2008).

[Petrenko, M. et al. 2008] Petrenko, M., Rajlich, V. and Vanciu, R. Partial Domain Comprehension in Software Evolution and Maintenance. In *The 16th IEEE International Conference on Program Comprehension (ICPC 2008)*, Amsterdam, The Netherlands (2008), 13-22.

[Pirró, G. and Euzenat, J. 2010] Pirró, G. and Euzenat, J. A feature and information theoretic framework for semantic similarity and relatedness. In *The 9th international semantic web conference (ISWC 2010)*, Shanghai, China (2010), 615-630.

[Pohl, K. et al. 2005] Pohl, K., Böckle, G. and van Der Linden, F. Software Product Line Engineering: Foundations, Principles, and Techniques, Springer (2005).

[Porter, M.F. 1980] Porter, M.F. An algorithm for suffix stripping. *Program*, 14, 3 (1980), 130-137.

[Rada, R. et al. 1989] Rada, R., Mili, H., Bicknell, E. and Blettner, M. Development and application of a metric on semantic nets. *IEEE Transactions on Systems, Man and Cybernetics*, 19, 1 (1989), 17-30.

[Rajlich, V. 2009] Rajlich, V. Intensions are a key to program comprehension. In *The IEEE 17th International Conference on Program Comprehension (ICPC '09)*, Vancouver, British Columbia, Canada (2009), 1-9.

[Rashid, A. et al. 2011] Rashid, A., Royer, J.C. and Rummler, A. Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way, Cambridge University Press (2011).

[Ratiu, D. and Deissenboeck, F. 2006] Ratiu, D. and Deissenboeck, F. How Programs Represent Reality (and how they don't). In *13th Working Conference on Reverse Engineering (WCRE '06)* Benevento, Italy (2006), 83-92.

[Ratiu, D. and Deissenboeck, F. 2007] Ratiu, D. and Deissenboeck, F. From Reality to Programs and (Not Quite) Back Again. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, Banff, Alberta, Canada (2007), 91-102.

[Ratiu, D. et al. 2008] Ratiu, D., Feilkas, M. and Jurjens, J. Extracting Domain Ontologies from Domain Specific APIs. In *12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, Athens, Greece (2008), 203-212.

[Reinhartz-Berger, I. et al. 2011] Reinhartz-Berger, I., Sturm, A. and Wand, Y. External Variability of Software: Classification and Ontological Foundations. In *International Conference on Conceptual Modeling - ER 2011*, Brussels, Belgium (2011), 275-289.

[Resnik, P. 1995] Resnik, P. Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, Montreal, Quebec, Canada (1995), 448-453.

[Resnik, P. 1999] Resnik, P. Semantic Similarity in a Taxonomy: An Information-Based Measure and its Application to Problems of Ambiguity in Natural Language. *Journal of Artificial Intelligence Research (JAIR)*, 11 (1999), 95-130.

[Rijsbergen, C.J.V. 1979] Rijsbergen, C.J.V. Information Retrieval, Butterworth-Heinemann (1979).

[Rokach, L. and Maimon, O. 2008] Rokach, L. and Maimon, O. Data Mining with Decision Trees: Theory and Applications, World Scientific (2008).

[Rugaber, S. 1995] Rugaber, S. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35, 20 (1995), 341-368.

[Ryssel, U. et al. 2011] Ryssel, U., Ploennigs, J. and Kabitzsch, K. Extraction of feature models from formal contexts. In *Proceedings of the 15th International Software Product Line Conference Workshops*, Munich, Germany (2011), 1-8.

[Ryssel, U. et al. 2012] Ryssel, U., Ploennigs, J. and Kabitzsch, K. Automatic library migration for the generation of hardware-in-the-loop models. *Science of Computer Programming*, 77, 2 (2012), 83-95.

[Salton, G. and Buckley, C. 1988] Salton, G. and Buckley, C. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24, 5 (1988), 513-523.

[Schlicker, A. et al. 2006] Schlicker, A., Domingues, F.S., Rahnenführer, J. and Lengauer, T. A new measure for functional similarity of gene products based on Gene Ontology. *BMC bioinformatics*, 7, 1 (2006).

[Seco, N. et al. 2004] Seco, N., Veale, T. and Hayes, J. An intrinsic information content metric for semantic similarity in WordNet. In *The 16th Eureopean Conference on Artificial Intelligence (ECAI'2004)*, Valencia, Spain (2004).

[Shannon, C.E. 1948] Shannon, C.E. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27, 3 (1948), 379-423.

[She, S. et al. 2011] She, S., Lotufo, R., Berger, T., Wasowski, A. and Czarnecki, K. Reverse Engineering Feature Models. In *The 33rd International Conference on Software Engineering, ICSE 2011*, Honolulu, USA (2011), 461-470.

[Simonyi, C. 1999] Simonyi, C. (1999). "Hungarian Notation." Retrieved June, (2016), from https://msdn.microsoft.com/en-us/library/aa260976%28v=vs.60%29.aspx.

[Slimani, T. 2013] Slimani, T. Description and evaluation of semantic similarity measures approaches. *International Journal of Computer Applications*, 80, 10 (2013).

[Slimani, T. et al. 2007] Slimani, T., Yaghlane, B.B. and Mellouli, K. Une extension de mesure de similarité entre les concepts d'une ontologie. In *4rth International Conference on Sciences of Electronic, Technologies of Information and Telecommunications (SETIT' 07)*, Tunisia (2007).

[Stroulia, E. and Systä, T. 2002] Stroulia, E. and Systä, T. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review*, 10, 1 (2002), 8-17.

[Studer, R. et al. 1998] Studer, R., Benjamins, V.R. and Fensel, D. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25, 1–2 (1998), 161-197.

[Sureka, V. and Punitha, S.C. 2012] Sureka, V. and Punitha, S.C. Approaches to Ontology Based Algorithms for Clustering Text Documents. *International Journal of Computer Technology and Applications*, 3, 5 (2012).

[Svahnberg, M. and Bosch, J. 2000] Svahnberg, M. and Bosch, J. Issues Concerning Variability in Software Product Lines. In *The International Workshop on Software Architectures for Product Families (IW-SAPF-3)*, Las Palmas de Gran Canaria, Spain (2000), 146-157.

[Tchechmedjiev, A. 2012] Tchechmedjiev, A. État de l'art: mesures de similarité sémantique locales et algorithmes globaux pour la désambiguïsation lexicale à base de connaissances. In *Conférence conjointe JEP-TALN-RECITAL*, France (2012), 295-308.

[Tizzei, L.P. et al. 2011] Tizzei, L.P., Dias, M., Rubira, C.M.F., Garcia, A. and Lee, J. Components meet aspects: Assessing design stability of a software product line. *Information and Software Technology*, 53, 2 (2011), 121-136.

[Tonella, P. and Potrich, A. 2007] Tonella, P. and Potrich, A. Reverse Engineering of Object Oriented Code, Springer-Verlag New York (2007).

[Vallet, D. et al. 2005] Vallet, D., Fernández, M. and Castells, P. An ontology-based information retrieval model. In *Second European Semantic Web Conference (ESWC 2005)*, Heraklion, Crete, Greece (2005), 455-470.

[van der Linden, F. 2002] van der Linden, F. Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, 19, 4 (2002), 41-49.

[van der Linden, F. and Müller, J.K. 1995] van der Linden, F. and Müller, J.K. Creating Architectures with Building Blocks. *IEEE Software*, 12, 6 (1995), 51-60.

[van der Linden, F. et al. 2007] van der Linden, F., Schmid, K. and Rommes, E. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer (2007).

[Van Deursen, A. and Klint, P. 2002] Van Deursen, A. and Klint, P. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10, 1 (2002).

[Von Mayrhauser, A. and Vans, A.M. 1995] Von Mayrhauser, A. and Vans, A.M. Program comprehension during software maintenance and evolution. *Computer*, 28, 8 (1995), 44-55.

[Warshall, S. 1962] Warshall, S. A Theorem on Boolean Matrices. *Journal of the ACM (JACM)*, 9, 1 (1962), 11-12.

[Weiss, D. 1998] Weiss, D. Commonality analysis: A systematic process for defining families. *Development and Evolution of Software Architectures for Product Families* (1998), 214-222.

[Weiss, D.M. and Lai, C.T.R. 1999] Weiss, D.M. and Lai, C.T.R. Software product-line engineering: a family-based software development process, Addison-Wesley Longman Publishing Co., Inc. (1999).

[Wiese, A. et al. 2011] Wiese, A., Ho, V. and Hill, E. A comparison of stemmers on source code identifiers for software search. In *the 27th IEEE International Conference on Software Maintenance (ICSM)*, Williamsburg, VA, USA (2011), 496-499.

[Wu, Z. and Palmer, M. 1994] Wu, Z. and Palmer, M. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, Las Cruces, New Mexico (1994), 133-138.

[Young, T.J. 2005] Young, T.J. Using aspectj to build a software product line for mobile devices, Master Thesis, The University of British Columbia (2005).

[Ziadi, T. et al. 2012] Ziadi, T., Frias, L., da Silva, M.A.A. and Ziane, M. Feature Identification from the Source Code of Product Variants. In *16th European Conference on Software Maintenance and Reengineering*, Szeged, Hungary (2012), 417-422.