Une approche métaheuristique pour la restructuration automatique des classes dans les applications logicielles orientées objet

par

Sabrine BOUKHARATA

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE LOGICIEL
M. Sc. A.

MONTRÉAL, LE 20 DÉCEMBRE 2019

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Ali Ouni, directeur de mémoire
Département de génie logiciel et des TI, École de Technologie Supérieure

M. Chamseddine Talhi, président du jury
Département de génie logiciel et des TI, École de Technologie Supérieure

M. Alain April, examinateur
Département de génie logiciel et des TI, École de Technologie Supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE "19 DÉCEMBRE 2019"

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je voudrais remercier mon directeur de recherche, Monsieur Ali Ouni, de m'avoir donné la possibilité de travailler avec lui et qui, par ses encouragements, ses remarques pertinentes et son suivi permanent a su accompagner et diriger le présent travail. Qu'il trouve ici l'expression de ma profonde gratitude.

Je voudrais remercier le professeur Alain April d'avoir accepté d'être mon examinateur et d'avoir fourni des efforts pour réviser mon mémoire.

Mes vifs remerciements s'adressent au monsieur le président de jury Chamseddine Talhi, pour l'honneur qu'il m'a fait en acceptant de juger mon travail.

Je voudrais remercier aussi, le ministère de l'Enseignement Supérieur et de la Recherche Scientifique de la Tunisie pour la confiance qu'ils m'ont attribués en finançant mes recherches grâce à une bourse d'excellence.

Enfin, je voudrais dédier ce travail en témoignage de mon grand amour à mes très Chers Parents pour ses sacrifices, mon mari pour son soutien continu, ses aides précieuses, ma sœur et mon frère pour leurs encouragements incessants et tous mes amis, en souvenir des bons moments que nous avons passés ensemble.

Une approche métaheuristique pour la restructuration automatique des classes dans les applications logicielles orientées objet

Sabrine BOUKHARATA

RÉSUMÉ

Les systèmes logiciels évoluent rapidement tout au long de leur cycle de vie. Les développeurs ne cessent d'ajouter de nouvelles fonctionnalités pour satisfaire les besoins des clients ou corriger des erreurs. Cependant, ces activités accroissent la complexité du logiciel et entraînent souvent des effets secondaires connus sous le nom de "code smells" ou mauvaises pratiques de programmation. Les mauvaises pratiques de code, assez fréquentes, sont caractérisées par de trop grandes classes (ou Blob) qui représentent une mauvaise pratique de conception ou d'implémentation de la part des développeurs. En effet, les développeurs placent souvent des méthodes sémantiquement non liées dans une seule et grande classe. Cette mauvaise pratique aboutit généralement à un logiciel difficile à comprendre, à entretenir et à faire évoluer, offrant parfois même une performance réduite.

Pour améliorer la qualité des logiciels, la technique la plus utilisée est le refactoring qui consiste à modifier la structure interne du code source tout en préservant son comportement externe. Pour corriger ces Blobs, l'opération de refactoring recommandée est d'effectuer un 'Extract class'. Elle permet de découper une trop grosse classe en des classes plus cohésives implémentant une seule fonctionnalité.

La plupart des travaux antérieurs ont examiné les relations structurelles entre les entités des classes pour générer des recommandations d'extraction des classes. D'autres travaux antérieurs ont considéré la similarité sémantique entre les méthodes pour assister l'extraction des classes. Cependant, tous ces travaux ignorent l'historique des changements du code appliqués dans le passé. De plus, ces approches présentent des approches déterministes. Malgré qu'une recherche déterministe puisse trouver de bonnes solutions, ces solutions sont souvent loin d'être optimales. Puisque de nombreuses combinaisons de membres des classes sont possibles, le processus de génération de recommandations peut être vu comme un problème d'optimisation.

Pour contourner les limites des solutions actuelles, nous avons développé une approche, intitulée BlobBreak, qui vise à automatiser l'opération de restructuration des classes. Nous avons utilisé un algorithme d'optimisation multi-objectif, le NSGA-II, pour trouver les combinaisons des méthodes et attributs à extraire dans ces nouvelles classes. Notre approche combine différents aspects portant sur les dépendances statiques et sémantiques ainsi que l'historique des changements du code source. Ainsi, les solutions générées doivent établir un compromis entre deux objectifs : (i) maximiser la cohésion des classes extraites et (ii) minimiser les dépendances entre les classes extraites.

Pour évaluer cette proposition, nous avons utilisé cinq projets du domaine du logiciel libre de moyenne et grande taille. Les résultats montrent que BlobBreak améliore la cohésion des trop

grandes classes ainsi que la qualité du logiciel. Les performances de BlobBreak surpassent celles d'une approche existante avec en moyenne, 77.99% de précision et 78.82% de rappel.

Mots-clés: Extract class refactoring, classes de trop grande taille, dépendance structurelle, dépendance sémantique, historique de changement de code

A meta-heuristic approach for automatic class restructuring in object-oriented software applications

Sabrine BOUKHARATA

ABSTRACT

Software systems evolve during their life cycle. Developers frequently add new features to satisfy customer needs or to correct some errors. However, these activities can increase system complexity and often cause code smells. Among these code smells, we find the God Class or **Blob**, which is a result of developers' bad practices. Indeed, they place semantically unrelated methods in one single class which results in a complex and non-cohesive one. This can therefore cause problems of comprehensibility and maintainability for the class itself and the whole software.

To improve the quality of software, the most used technique is refactoring, which consists of modifying its internal structure while preserving its external behavior. To fix blobs, the generally used refactoring operation is **Extract class** that allows the developer to split a large class into more cohesive ones implementing a single feature.

To propose recommendations for class extraction, most of previous works have examined structural relationships between the class entities, while others have considered semantic similarity between methods. However, all these works ignore the history of code changes applied in the past. Additionally, all these researches present deterministic approaches that are inadequate to the problem due to the huge space of all the possible combinations of class members. Hence, the recommendation process is inherently an optimization problem.

To deal with these limitations, we have developed an approach, named **BlobBreak**, which focuses on automating the Extract Class Refactoring operation using a meta-heuristic technique. We used a multi-objective algorithm to find the combinations of entities (methods and attributes) and extract them in new classes. Our approach combines different aspects of static and semantic dependencies as well as the history of code change. The solutions generated must be able to find a compromise between two objectives to (i) maximize the cohesion of the extracted classes and (ii) minimize the dependencies between the system classes.

To evaluate our approach, we considered five Open Source projects with medium and large sizes. The obtained results show that BlobBreak improves the cohesion of the extracted classes as well as the quality of the studied projects. Additionally, the evaluation of BlobBreak's efficiency showed that our approach outperforms JDeodorant,an existing approach, with a precision and recall of 78% and 77% respectively.

Keywords: Extract class refactoring, God class, structural dependency, semantic dependency, history of code change

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

AG	Algorithme Génétique
CAMC	Cohesion Among Methods Of a Class
CBO	Coupling Between Objects
CDM	Call Based Dependency Between Methods
CSM	Conceptual Similarity between Methods
MOEA	Multiobjectifs Evolutifs Algorithms
NSGA	Non dominated sorting genetic algorithm
JDT	Java Development Tools
KLOC	kilo of lines of code
LCOM	Lack of cohesion of methods
SBSE	Serch based software engineering
SCV	Système de Contrôle de Version
SPEA	Strength Pareto Evolutionary Algorithm
PRU	Principe de Responsabilité Unique
SSM	Structuaral Similarity Between Methods

INTRODUCTION

Contexte de la recherche

Plusieurs études ont démontré que les tâches de maintenance logicielle peuvent représenter jusqu'à 75% du coût total de développement d'un logiciel (Mealy, Carrington, Strooper & Wyeth, 2007). La maintenance logicielle est la modification d'un produit logiciel après sa livraison telle que la correction des erreurs de conception, l'amélioration des fonctionnalités existantes et la modification d'une conception pour tenir compte des nouvelles exigences. En effet, les modifications logicielles sont inévitables pour les systèmes logiciels modernes. Il est difficile d'estimer les conséquences potentielles d'un changement après la modification de certaines parties du logiciel. Au fur et à mesure que ces changements s'accumulent, le code devient volumineux et complexe.L'utilisation de mauvaises pratiques lors de ces changements peuvent introduire des « odeurs de code » (de l'anglais Code Smells) dans le code source du logiciel. Une odeur de code fait référence à une mauvaise pratique de programmation introduite lors des phases de conception ou de mise en œuvre du code source d'un programme. Généralement, les odeurs de code indiquent l'existence de mauvaises pratiques de programmation ou de conception qui peuvent nuire au développement du logiciel. Pour maîtriser les coûts de maintenance, des efforts importants doivent être consacrés à l'élimination de ces mauvaises pratiques pour s'assurer d'une bonne qualité de code et s'assurer de conserver la conception en bon état. Conséquemment, les développeurs et les mainteneurs doivent corriger ces odeurs de code pour améliorer la qualité du logiciel et par conséquent sa maintenabilité et son évolution future.

Problématique

Une des mauvaises pratiques de code est le Blob ou la trop grande taille d'une classe. Cette mauvaise pratique de programmation se localise dans des fragments de code où une classe de grande taille monopolise le comportement d'un logiciel, tandis que d'autres classes contiennent

principalement les données. Un Blob peut contenir des responsabilités qui se chevauchent avec d'autres modules du logiciel, se donnant ainsi les caractéristiques d'une conception procédurale. Ces caractéristiques rendent les Blobs difficiles à maintenir et susceptibles aux défauts. Par conséquent, il est nécessaire d'identifier et de signaler l'existence de telles odeurs de code et de suggérer des mesures correctives pour les éliminer. En effet, les meilleures pratiques de conception des classes suggèrent de séparer les responsabilités d'une classe de trop grande taille dans des classes plus petites et cohésives et ceci pour améliorer la structure de code. Celle-ci fait référence au rapport étroit qui existe entre les opérations de la même classe. Une cohésion élevée signifie que les membres de la classe sont fortement liés, car ils opèrent sur la même abstraction.

Dans ce contexte, de nombreuses méthodes ont été proposées pour corriger le Blob. L'une des techniques largement utilisées est le refactoring *Extract Class* qui sert à découper une classe de trop grande taille en des classes de taille appropriée pour améliorer la qualité de la classe en termes de cohésion. L'idée est de déléguer certaines responsabilités d'un Blob à de nouvelles classes. Cette technique est utilisée pour améliorer différents aspects de la qualité de la classe tels que la réutilisabilité, la maintenabilité et la complexité.

Vu la grande complexité des Blobs et des systèmes logiciels de grande taille, il est difficile de s'appuyer sur l'expertise humaine seule pour effectuer un *Extract Class*. Par conséquent, il est nécessaire de créer des outils automatisés pour corriger ces Blobs. Ainsi, puisque de nombreuses combinaisons d'entités de classes (c.-à-d. méthodes et attributs) sont possibles, une recherche déterministe peut ne pas être efficace pour résoudre ce type de problèmes. Les travaux existants, bien qu'ils proposent en majorité des outils automatisés, proposent tous des approches déterministes. En plus, ces approches se basent uniquement sur les dépendances structurelles ou la similarité sémantique entre les membres des classes et négligent l'historique des changements du code appliqués dans le passé. Cette heuristique est un aspect qui pourrait être exploité pour améliorer l'exactitude des nouvelles solutions de recommandation de *refactoring Extract Class*.

Contributions

Le problème précédemment identifié nous a motivé à proposer une nouvelle approche basée sur la recherche méta-heuristique, intitulée BlobBreak. Pour automatiser le processus de restructuration des classes dans les applications logicielles orientées objet. Puisque le problème de restructuration des classes est un problème d'optimisation combinatoire, nous avons utilisé un algorithme génétique multi objectif, NSGA-II publié par Deb (Deb, Pratap, Agarwal & Meyarivan, 2002), pour générer automatiquement les meilleures recommandations de restructuration des classes. Les deux objectifs que nous avons définis sont : (i) maximiser la cohésion des classes extraites et (ii) minimiser les dépendances entre les classes de système. Afin de déterminer la force des relations entre les méthodes, nous avons utilisé trois différents heuristiques : les dépendances structurelles, les dépendances sémantiques et le cochangement. Pour évaluer cette proposition de solution, nous avons utilisé cinq projets du domaine du logiciel libre de moyenne et grande taille. À la suite de l'expérimentation, l'approche proposée améliore considérablement la cohésion de classes restructurées et avec des performances qui surpassent une approche existante (JDeodorant).

Organisation du mémoire

Ce rapport est structuré selon le plan suivant. Les concepts de base ainsi qu'une étude des travaux antérieurs sont présentés dans le premier chapitre afin de positionner les contributions de notre travail par rapport aux solutions existantes. L'idée principale de l'approche proposée est présentée dans le deuxième chapitre où les différentes phases de sa conception sont détaillées. L'expérimentation de la solution proposée est décrite dans le troisième chapitre. Enfin, nous concluons le travail par la présentation des apports et des limitations de cette recherche ainsi qu'une discussion sur les travaux futurs possibles.

CHAPITRE 1

ÉTAT DE L'ART

1.1 Introduction

Ce chapitre propose une revue de la littérature sur les travaux de recherche liés à notre sujet. Nous fournissons d'abord les définitions des concepts de base nécessaires à la compréhension de ce mémoire. Nous examinons ensuite les travaux correspondants en rapport avec les thèmes principaux de ce travail de recherche. Ces travaux sont divisés en deux parties : (1) les approches de refactoring de façon générale et (2) les approches spécifiques à l'extraction de classe.

1.2 Concepts de base

1.2.1 Odeurs de code

Une odeur de code est un indicateur qui reflète la mauvaise structure du code qui viole les principes fondamentaux des meilleures pratiques de conception logicielle (Fowler, Beck, Brant, Opdyke & Roberts, 1999). Les odeurs de code sont souvent introduites involontairement ou par manque d'expertise par les développeurs lors de la conception initiale ou lors du développement et de la maintenance du logiciel. Cependant, il est à noter que ces odeurs de code ne doivent pas être confondues avec les erreurs de compilation. Fowler et al (Fowler *et al.*, 1999) ont présenté dans leur livre un catalogue de 22 odeurs de code incluant le code dupliqué, les longues méthodes et les classes de grande taille.

Fowler (Fowler, 2018) a souligné l'impact négatif des odeurs de code sur la qualité des logiciels. L'identification des odeurs de code est une activité populaire et amplement publiée. À cette fin, plusieurs approches pour l'identification et la correction d'odeurs de code ont été proposées et développées.

1.2.2 Classe de trop grandes tailles ou Blob

Le Blob est l'une des odeurs de code les plus connues qui se trouvent dans des systèmes logiciels où une classe monopolise le comportement du système (Brown, Malveau, McCormick & Mowbray, 1998), tandis que d'autres classes encapsulent principalement les données. Cette odeur de code est caractérisée par un fragment d'un diagramme de classes composé d'une seule classe de contrôle complexe entourée par autres simples classes de données. Le problème-clé est que la majorité des responsabilités sont attribuées à une seule classe. En principe, une classe devrait disposer un seul concept et ne devrait changer que lorsque ce concept se développe (Martin & Martin, 2006). La violation de ce principe résulte en la présence d'une classe complexe et moins cohésive (Brown *et al.*, 1998).

Le Blob s'accompagne souvent d'un code redondant, ce qui rend la différenciation entre les fonctionnalités utiles de la classe de Blob et le code non utile très difficile. En plus, cette odeur de code crée un couplage étroit, un manque de cohésion entre les attributs et des opérations encapsulées dans une seule classe ainsi qu'une absence de conception orientée objet.

Généralement, il y a deux types de classes : certaines détiennent beaucoup de données du système (c.-à-d. « Data Classes » ou « Lazy Classes ») et d'autres mettent en œuvre une grande partie de la fonctionnalité du système en termes de plusieurs méthodes (c.-à-d. « Behavioral God Classes »). Dans le premier cas, les développeurs peuvent redistribuer les attributs de la classe. Dans le second cas, ils peuvent utiliser un refactoring intitulé « Extract Class ».

1.2.3 Refactoring

Le concept de refactoring a été d'abord introduit par Opdyke en 1992 (Opdyke, 1992) comme une activité qui aide les développeurs à corriger les odeurs de code et à réduire la complexité croissante d'un système logiciel. Dans son livre, Fowler (Fowler *et al.*, 1999) définit le refactoring de code comme une technique qui permet de modifier la structure interne de code sans changer son comportement externe dans le but de le rendre plus simple et plus lisible. Parce que l'idée principale est de faciliter l'utilisation et la compréhension du logiciel et d'améliorer les différents

aspects de la qualité logicielle tels que la maintenabilité, l'extensibilité et la réutilisabilité (Fowler *et al.*, 1999).

Le refactoring consiste à améliorer le code via différentes techniques comme l'ajout de la documentation manquante, la suppression du code dupliqué ou obsolète et la restructuration des classes. Il existe environ 72 opérations de refactoring qui sont cataloguées par Fowler et al (Fowler *et al.*, 1999). Parmi ces opérations, on distingue les changements les plus simples comme *Extract Local Variable*, *Move Method*, *Extract Method* et des changements les plus compliqués tels que *Extract Class*.

1.2.4 Extract Class Refactoring

En raison de la grande complexité des Blobs, il pourrait être difficile et fastidieux de reconnaitre la bonne décomposition manuellement. Nous avons vu que ces classes doivent être redéfinies pour améliorer leur conception interne et donc leur maintenabilité. Le refactoring *Extract Class* a été définit pour corriger cette odeur de code en déléguant certaines responsabilités à de nouvelles classes. Plus précisément, le refactoring Extract class (Brown *et al.*, 1998) est une technique qui permette de diviser les classes avec nombreuses responsabilités dans différentes classes plus cohésives. Un inconvénient de diviser une classe en plusieurs classes manuellement est que les nouvelles classes devront passer des messages entre elles, alors que dans le Blob d'origine, ces messages étaient à transmettre entre les méthodes de la même classe. Le couplage ajouté induit une dépendance entre les classes, c'est-à-dire que changer une classe individuellement apporte un changement indésirable dans une autre classe. Une solution souhaitable à ce problème sera d'appliquer la meilleure technique de restructuration de classe qui divise une classe en plusieurs classes cohérentes avec un minimum d'assistance de développeur et qui maintient le couplage entre les classes extraites au minimum.

1.2.5 Le génie logiciel basé sur les métaheuristiques (SBSE)

SBSE (Search Based Software Engineering) consiste en l'application des algorithmes d'optimisation métaheuristiques pour résoudre des problèmes en génie logiciel (Harman & Jones, 2001). En utilisant le SBSE, une tâche de génie logiciel est formulée comme un problème de recherche métaheuristique en définissant la solution candidate et une fonction d'évaluation qui guide vers la solution optimale.

L'application des techniques de SBSE est grandissante en génie logiciel, car ils fournissent un moyen pour résoudre des problèmes difficiles qui impliquent des objectifs en conflit en utilisant une approche automatisée. Parmi les motivations qui encouragent les développeurs à utiliser SBSE est que la plupart des questions en génie logiciel sont souvent dans un langage qui réclame simplement une solution basée sur l'optimisation.

La SBSE est devenue un domaine de recherche et de pratique en pleine croissance. Ces techniques sont très utilisées dans le milieu académique ainsi que dans l'industrie (Harman, Mansouri & Zhang, 2012). En effet, des nombreuses approches SBSE ont été appliquées à une grande variété de problèmes de génie logiciel, y compris les tests de logiciel (McMinn, 2004), la maintenance logicielle (Le Goues, Nguyen, Forrest & Weimer, 2011), la gestion de projet (Alba & Chicano, 2005) et le refactoring (Harman & Tratt, 2007).

1.2.6 Algorithmes d'optimisation

Les ingénieurs affrontent quotidiennement des problèmes qui sont d'une complexité grandissante et, qui apparaissent dans plusieurs secteurs techniques. Le problème à résoudre peut fréquemment être énoncé sous la forme d'un problème d'optimisation.

Ce type de problème est caractérisé par (Lachance, 2014) :

- un espace de recherche,
- Une ou plusieurs fonctions à optimiser : Fonction objectif (mono-objectif ou multiobjectif),

- un ensemble de contraintes à considérer.

1.2.6.1 Optimisation mono-objectif

Un problème d'optimisation s'appelle mono-objectif, c'est lorsqu'un seul objectif est défini. Cet objectif est soit à minimiser ou à maximiser par rapport à certains paramètres. Ces paramètres s'appellent variables d'optimisation. Dans ce cas la solution optimale est clairement connue, c'est celle qui a le coût optimal (minimal ou maximal).

La définition claire de ce problème est, à chaque instance de ce dernier est associé un ensemble Q des solutions admissibles respectant des contraintes spécifiques. Soit f une fonction permettant d'évaluer chaque solution admissible. Résoudre l'instance (Q) du problème d'optimisation permet de trouver la solution optimale s* de Q qui minimise ou maximise la valeur de la fonction fitness (Lachance, 2014).

Parmi les algorithmes d'optimisation mono-objectif les plus utilisés, il existe l'algorithme génétique.

1.2.6.1.1 Algorithme génétique

L'algorithme génétique (AG) (Holland & Goldberg, 1988) est un algorithme méta-heuristique qui a été introduit par John Holland. Il imite le processus de variation génétique et de sélection naturelle et résume la théorie développée par Charles Darwin (Holland & Goldberg, 1988). Le principe de résolution des problèmes d'optimisation par les algorithmes génétiques est illustré dans la Figure 1.1. Initialement, une population de N individus (chromosomes pour faire le lien avec la biologie) qui représente des solutions réalisables du problème est générée aléatoirement.

Souvent les individus sont codés pour des raisons de manipulation (croisement ou mutation) et les codes utilisés varient en fonction du problème traité.

L'étape suivante est la sélection des meilleurs chromosomes qui participeront à l'établissement de la génération future. Une mauvaise sélection peut conduire à une convergence trop rapide

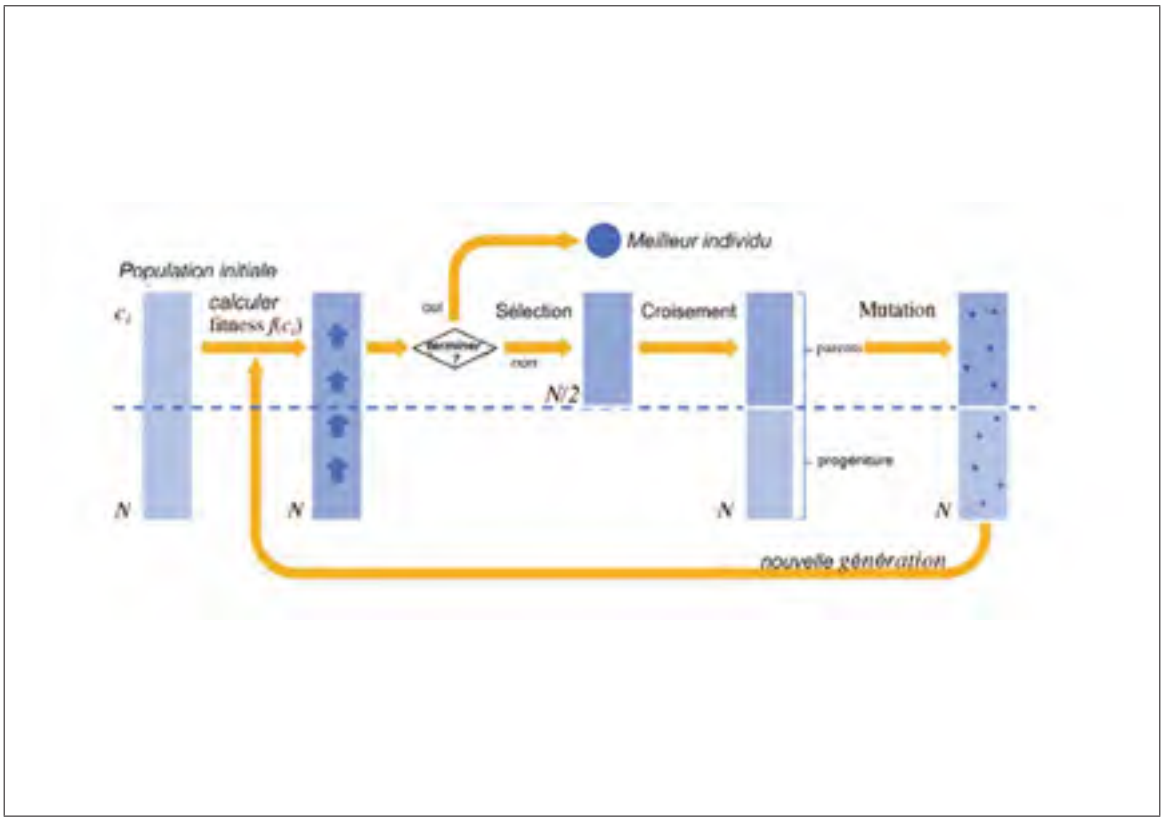

Figure 1.1 Principe général des algorithmes génétiques

vers des optimums locaux qui peuvent s'avérer être très éloignés de la solution. Ainsi la sélection est souvent précédée d'une évaluation des chromosomes résultats via la fonction objectif (fitness f (ci) dans la Figure 1.1).

Chaque chromosome choisi est ensuite soumis à une probabilité de croisement Pc avec un autre engendrant ainsi deux nouveaux chromosomes qui ont hérité d'une partie du code génétique de leurs parents, et aussi d'une probabilité de mutation Pm pour une diversité plus large de la nouvelle population (et couvrir plus de solutions potentielles).

Les étapes décrites précédemment se succèdent dans cet ordre de génération en génération, le critère d'arrêt peut dépendre, par exemple, du nombre de générations ou la convergence de la population vers une même solution (homogénéisation de la population).

Dans les paragraphes suivants, nous expliquons les paramètres utilisés dans l'algorithme génétique et ses différents procédés.

1. Codage des individus :

Il est évident que l'algorithme dépendra très fortement du type de codage choisi, mais cela revient en fait à la nature du problème traité. Une solution (ou individu) est donc un chromosome formé d'une (ou plusieurs) chaines de bits où chaque bit représente une gêne et son contenu appelé allèle. Nous décrivons dans ce qui suit les codages les plus connus (Holland & Goldberg, 1988) à savoir le codage binaire, le codage réel, le codage à caractères multiples et le codage d'arbre.

- **Codage binaire :** Le principe de ce codage consiste à coder la solution selon une séquence de bits. Ces chaînes de bits sont, ensuite, concaténées l'une après l'autre pour former une grande séquence de bits qui représente le chromosome. Ce type de codage est le plus utilisé ;
- **Codage réel :** Chaque chromosome est une chaîne de valeurs entières réelles ;
- **Codage à caractères multiples :** Chaque chromosome est une chaîne de valeurs et les valeurs peuvent être liées à des problèmes, des numéros de forme, des nombres réels ou des caractères à certains objets complexes ;
- **Codage d'arbre :** Ce codage utilise une structure arborescente pour coder les solutions.

2. La fonction objectif (fitness) :

La fonction objectif permet à partir des chromosomes de quantifier numériquement la validité de la solution qu'elle représente et ainsi de mesurer le degré d'adaptation d'un individu à son environnement (Holland & Goldberg, 1988). Cette fonction varie d'un problème à un autre. Elle varie aussi si on essaie de minimiser ou de maximiser un problème.

À chaque nouvel individu créé, nous associons une valeur (appelée fitness, valeur d'adaptation ou valeur d'évaluation) qui mesure la qualité de l'individu et qui représente la performance de

l'individu vis-à-vis au problème à résoudre. Nous obtenons donc une valeur de fonction objectif pour chaque individu. Si la population est de taille n, nous allons avoir alors n valeurs au total pour l'ensemble de la population. L'efficacité d'un AG s'appuie pour une grande partie, sur la qualité de cette fonction. Cependant, aucune règle n'existe pour définir cette fonction.

3. La sélection :

Cet opérateur permet de définir quels sont les individus de la population initiale qui vont être dupliqués dans la nouvelle population et vont servir de parents (Holland & Goldberg, 1988). Il est fondé sur la performance des individus, estimée à l'aide de la fonction objectif précédemment décrite.

L'implantation de cet opérateur peut se faire par plusieurs méthodes. On cite ci-dessous quelques-unes extraites de (Holland & Goldberg, 1988) :

- **Sélection par roulette :** Elle consiste à attribuer à chaque individu un secteur dont l'angle est proportionnel à sa performance. Ces secteurs sont, ensuite, concaténés sur un axe gradué. On fait tourner la roue d'une manière aléatoire et quand elle cesse de tourner, on sélectionne l'individu correspondant au secteur désigné ;
- **Sélection par tournoi :** Cette méthode consiste à choisir aléatoirement un sous-ensemble d'individus dans la population, puis à sélectionner le meilleur individu. En général, si T est la taille du tournoi, on sélectionne aléatoirement T individus dans la population et on les fait entrer en compétition, seul le vainqueur est retenu. Le nombre de participants à un tournoi T, appelé la taille du tournoi, est utilisé pour faire varier la pression de cette sélection. Si ce nombre est élevé, alors la pression sera forte et les faibles individus auront une petite chance d'être choisis. La méthode de tournoi la plus commune est le tournoi binaire, où on choisit deux individus aléatoirement (T =2) puis on sélectionne le meilleur. Cette méthode est jugée comme étant celle avec laquelle on obtient les résultats les plus satisfaisants (Holland & Goldberg, 1988) ;

- **Élitisme :** Cette méthode consiste à sélectionner les n meilleurs individus de la population P pour la reproduction. Cette méthode améliore considérablement les performances de l'AG car elle permet de ne pas perdre les meilleures solutions.

4. Les opérateurs génétiques :

Les opérateurs génétiques permettent la diversification de la population au cours des générations et la bonne exploration de l'espace de recherche représenté par l'espace des solutions (Holland & Goldberg, 1988). Dans ce qui suit, ces différents opérateurs sont présentés avec détail.

A) Le croisement :

C'est un opérateur qui consiste à combiner deux individus quelconques (dits parents) pour en ressortir deux autres individus (dits enfants) pas forcément meilleurs que les parents. L'objectif de cette opération est d'enrichir la diversité de la population et cela en combinant les gènes. Il existe plusieurs variantes de cet opérateur. Nous citons deux types de croisement.

- **Croisement en un point :** Cette méthode consiste à choisir aléatoirement une position inter-gènes (un point de croisement) pour chaque couple. On échange, ensuite, les deux sous-chaînes de chacun des chromosomes ce qui produit deux enfants. La Figure 1.2 illustre un croisement de point unique et on peut observer que les bits à côté du point de croisement sont échangés pour produire les enfants.

- **Croisement en deux point :** Cette méthode est une généralisation du croisement à un point. Elle permet de choisir aléatoirement deux points de croisement pour les deux parents. On échange, ensuite, les deux sous-chaines situées entre les deux points de croisement de chacun des chromosomes parents, ce qui produit deux nouveaux chromosomes enfants.

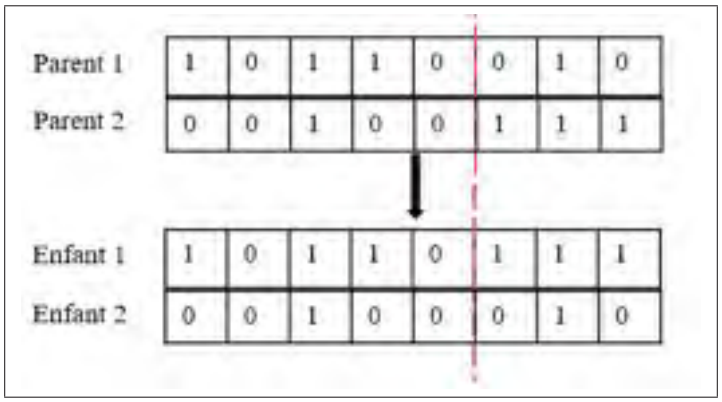

Figure 1.2 Exemple de croisement à un point

Dans la Figure 1.3 les lignes pointillées indiquent les points de croisement. Ainsi, le contenu entre ces points est échangé entre les parents pour produire de nouveaux enfants pour s'accoupler dans la prochaine génération.

Figure 1.3 Exemple de croisement à deux points

B) La mutation :

Le rôle de cet opérateur consiste à changer avec une certaine probabilité, la valeur allylique d'un gène (élément constituant le chromosome). Cet opérateur a pour but de garantir l'exploration de l'espace de recherche. Il existe de nombreuses formes de mutation pour les différents types de

représentation. Par exemple, pour la représentation binaire une simple mutation peut consister à inverser la valeur de chaque gène.

1.2.6.2 Optimisation multi-objectif

Généralement, plusieurs problèmes d'optimisation sont caractérisés par plus qu'un seul objectif. Ces objectifs sont dans la plupart de temps contradictoires. Prenant l'exemple d'un problème à trois objectifs, la solution optimale est choisie parmi plusieurs correspondants aux meilleurs compromis possible pour résoudre ce problème. Tandis que, pour un problème mono-objectif, la solution optimale est un seul point clairement défini (Lachance, 2014).

Les problèmes multi-objectifs sont plus difficiles à traiter que les problèmes mono-objectifs. En effet, la résolution de ces problèmes demeure difficile parce qu'aucune relation n'existe presque entre les solutions. Comme il était expliqué avant, dans ces problèmes, il existe n fonctions objectifs et m contraintes. La forme générale du problème est la suivante :

Trouver $x = [x_1, .., x_n]$
qui minimise/maximise $f(x) = \{f_1(x), ..., f_n(x)\}$
sujet à $g_j(x) \leq 0,\ j = 1, m$

Les solutions résultantes sont meilleures sur quelques objectifs et moins bonnes sur le reste. Donc il n'existe généralement pas une solution unique que nous pouvons dire que c'est la solution optimale pour tous les objectifs. Donc le principe ici est qu'aucune amélioration ne peut être faite sur un objectif sans dégrader au moins un autre objectif. Il est nécessaire, pour identifier les meilleurs points, de définir une relation d'ordre entre ces éléments. La plus célèbre est la relation de dominance. L'ensemble des meilleures solutions constitue le front Pareto.

- **Notion de dominance et de Pareto :**

Le fait que les objectifs dans un problème d'optimisation multi-objectif sont contradictoires. Nous trouvons généralement, un ensemble de solutions où une relation de dominance est définie entre ces derniers.

Le principe de base dans une approche Pareto (Pareto, 1964) est de trouver un compromis entre les objectifs de façon à améliorer un objectif et détériorer au moins un autre au même temps (Pareto, 1964). Pour sélectionner les solutions, les approches évolutives utilisent directement la notion de dominance. Ces solutions construisent le front de Pareto.

Prenant l'exemple de la Figure 1.4, x ne domine pas y, y ne domine pas x, mais tous les deux dominent le point z. Donc les x et y appartiennent au front de Pareto.

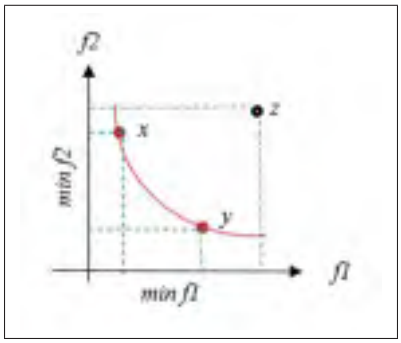

Figure 1.4 Front Pareto

Généralement, pour résoudre un problème multi-objectif, il faut passer par deux étapes :

- La recherche des solutions pour construire le front Pareto : C'est le but d'optimisation multi-objectif ;
- Le choix de la solution à admettre : C'est le décideur qui va choisir parmi les solutions trouver par l'algorithme, la solution qui résolu le problème.

Différents travaux de recherche ont appliqué les algorithmes évolutifs multi-objectifs (MOEA) présenté par (Coello, Lamont, Van Veldhuizen et al., 2007). Ces recherches ont montré que ces méthodes sont puissantes et robustes dans la résolution des problèmes multi objectifs, trois catégories de solutions ont été proposées : la somme pondérée des fonctions objectifs, les approches basées sur la population et celles basées sur le concept de Pareto.

Dans la dernière catégorie, nous trouvons différentes méthodes comme le ”Non-dominated Sorting Genetic Algorithm” (NSGA) proposé par Srinivas et Deb (Srinivas & Deb, 1994). En plus, Deb et ses collègues. (Deb *et al.*, 2002) ont développé le NSGA-II qui est la deuxième version de la NSGA. En 2013, Deb et al ont proposé NSGA-III (Deb & Jain, 2013) la troisième version. Autres méthodes comme SPEA2 ont été fondés afin de trouver la meilleure solution pour les problèmes multi-objectifs. Nous présentons dans la suite trois algorithmes génétique multi-objectifs les plus populaires.

1.2.6.2.1 NSGA-II :

NSGA-II (Deb *et al.*, 2002) est l'un des algorithmes d'optimisation multi-objectifs les plus populaires. L'idée est la suivante : faire développer une population de solutions candidates vers un ensemble de solutions quasi optimales. Comme décrit dans l'algorithme 1.1, la première étape de NSGA-II consiste à créer de manière aléatoire une population d'individus codés à l'aide d'une représentation spécifique (ligne 2). Ensuite, il évalue les fonctions objectifs et attribut un rang à chaque individu basé sur le tri non dominé (ligne 3 et 4). Après, une population d'enfants est générée à partir de la population de parents en utilisant des opérateurs génétiques tels que le croisement et la mutation (ligne 5). NSGA-II trie pour chaque génération les individus issus des populations parente et descendante, en tenant compte de la non-dominance, créant ainsi plusieurs fronts (lignes 8 et 9). Le premier front est composé de toutes les solutions non dominées. Le second front comporte les solutions dominées par une seule solution. Le troisième front est composé des solutions dominées par deux autres solutions, et les fronts sont créés jusqu'à ce que toutes les solutions soient classifiées. La distance d'encombrement calcule la distance qui sépare les voisins d'une solution donnée (ligne 10). Après le calcul, les solutions

sont triées par ordre décroissant. Les solutions aux limites de l'espace de recherche bénéficient de valeurs élevées de distance d'encombrement, car elles sont plus diversifiées, mais avec moins de voisins (ligne 13). Ensuite, le tournoi binaire sélectionne des individus de front inférieur et des nouvelles populations sont générées avec les opérateurs de recombinaison et de mutation (ligne 14). Elles s'appellent solutions non dominées ou front de Pareto. Ces solutions sont celles

Algorithme 1.1 Algorithme NSGA-II

```
1  Input : N, g, f_k(X)
2  Générer une population aléatoire - taille N;
3  Évaluer les valeurs des objectifs;
4  Attribuez un rang (niveau) basé sur le tri de Pareto;
5  Générer une population d'enfants (sélection du tournoi binaire, recombinaison et
     mutation);
6  for i ← 1, g do
7      for chaque parents et enfant dans la population do
8          Attribuer un classement (niveau) basé sur le tri par Pareto;
9          Générer des ensembles de solutions non dénommées;
10         Déterminer la distance d'encombrement;
11         Ajouter des solutions à la prochaine génération à partir du premier front;
12     end
13     Sélectionnez des points sur le bas du front avec une distance d'encombrement élevée;
14     Créer la prochaine génération (Sélection, recombinaison et mutation);
15 end
```

qui offrent un compromis approprié entre tous les objectifs sans en dégrader aucun.

1.2.6.2.2 NSGA-III :

NSGA-III (Deb & Jain, 2013) est une méthode d'optimisation qui fonctionne bien avec des problèmes ayant plusieurs objectifs (trois ou plus). C'est une extension de l'algorithme NSGA-II. Le cadre de base reste similaire à l'algorithme NSGA-II avec des changements significatifs dans son mécanisme de sélection. La principale différence entre les deux algorithmes est que NSGA-III utilise un ensemble de points de référence pour maintenir la diversité des points de Pareto pendant la recherche. Cela se traduit par une distribution très uniforme des points de Pareto, même lorsque le nombre d'objectifs est grand. Les points de référence choisis peuvent

être soit prédéfinis, soit fournis par l'utilisateur. En l'absence d'informations sur les préférences, tout placement structuré prédéfini peut être adopté, comme celui de Das et Dennis (Das & Dennis, 1998).

1.2.6.2.3 SPEA2 :

SPEA2 (Zitzler, Laumanns & Thiele, 2001) est un algorithme d'optimisation évolutif, multi objectifs. "Strength Pareto Evolutionary Algorithm" est une extension de l'algorithme génétique pour des problèmes d'optimisation multi-objectif.

L'objectif de l'algorithme est de localiser et de maintenir un front de solutions non dominées, idéalement un ensemble de solutions optimales de Pareto. Ceci est réalisé en utilisant un processus évolutif pour explorer l'espace de recherche, et un processus de sélection combinant le degré de domination d'une solution candidate et une estimation de la densité du front de Pareto. Une archive de l'ensemble non dominé est maintenue séparée de la population de solutions candidates utilisées dans le processus évolutif, fournissant une forme d'élitisme.

1.3 Travaux connexes

1.3.1 Approches de détection et correction des odeurs de code

Plusieurs études ont récemment porté sur la détection des odeurs de code dans un logiciel en utilisant différentes techniques. Ces techniques sont classées dans trois catégories : manuelles, semi-automatiques et automatiques.

1.3.1.1 Approches manuelles

Il existe des stratégies pour détecter et corriger les codes smells qui nécessite l'intervention de l'être humain, dans le livre de Fowler (Fowler *et al.*, 1999), il a proposé que les développeurs doivent détecter les odeurs de code d'une façon manuelle en spécifiant des techniques de ré usinage pour chaque type d'odeur de code, de leur part Travassos et ses collègues (Travassos,

Shull, Fredericks & Basili, 1999) ont également proposé une approche manuelle pour détecter et corriger les odeurs de code et c'est en fournissant une aide technique ou des conseils pratiques.

Alikacem et Sahraoui (Alikacem & Sahraoui, 2009) ont suggéré une méthodologie pour identifier les odeurs de code dans les systèmes orientés objet. La terminologie a fourni les lignes directrices à l'aide de logique floue et des métriques.

Ces approches présentent plusieurs inconvénients, en effet ces techniques demandent un effort des développeurs et une forte interprétation, ce qui peut générer des pertes de temps. Aussi parmi les limites de ces approches est que les résultats ne sont pas tout à fait exacts, pour contourner ces problèmes des techniques semi automatisées sont apparues.

1.3.1.2 Approches semi-automatiques

Plusieurs autres travaux de recherche basés sur les techniques de refactoring semi-automatique seront discutés dans cette section.

Moha et ses collègues (Moha, Gueheneuc, Duchien & Le Meur, 2009) ont présenté une méthode spécifique intitulée DÉCOR décrivant toutes les étapes nécessaires à la spécification et à la détection des odeurs de code et de design. Ils ont aussi développé, DETEX, un outil qui permet aux développeurs de spécifier des odeurs à un niveau d'abstraction élevé à l'aide d'un vocabulaire unifié et d'un langage spécifique à un domaine. Ce vocabulaire est obtenu à partir d'une analyse de domaine approfondie.

Feng et ses collègues (Feng, Zhang, Wang & Wang, 2004) ont indiqué que la présence des odeurs de code dans le système dégrade les performances du code source. Ils ont proposé une microarchitecture pour générer un modèle de conception basé sur XML.

Baudry et ses collègues (Baudry, Le Traon & Sunyé, 2004) ont avancé l'idée de reconnaître les odeurs de code au niveau de la conception plutôt qu'au niveau de la mise en œuvre. La méthode d'extension UML a été utilisée pour améliorer la conception.

Murphy-Hill et ses collègues (Murphy-Hill, Black, Dig & Parnin, 2008) ont proposé différents types de techniques de refactoring comme la manipulation des historiques de code, la compréhension d'anciens travaux des programmeurs et l'archivage des outils qui ont été utilisés. Zhang et ses collègues (Zhang, Baddoo, Wernick & Hall, 2011) ont effectué une relation entre les fautes dans les logiciels et six mauvaises odeurs données par Fowler et al (Fowler *et al.*, 1999). La relation justifiée sera utile dans l'application de refactoring par les développeurs.

Certains environnements de développement intégrés (IDE) modernes, tels que Eclipse [1] et Intellij [2] et plusieurs autres, fournissent un support semi-automatique de processus de refactoring. Ces outils permettent d'augmenter la vitesse avec laquelle les programmeurs peuvent maintenir le code tout en diminuant la probabilité qu'ils introduisent de nouveaux bogues.

La principale limite de ces approches est que les développeurs essayent d'appliquer des améliorations séparément sans considérer l'ensemble du programme. Aussi, l'approche semi-automatique est limitée à quelques opérations de refactoring possibles et peu de métriques de qualité.

1.3.1.3 Approches automatiques

Dans le cas de techniques de détection et de correction automatique, cadriciels (de l'anglais « frameworks »), des enquêtes et des outils entièrement automatiques ont été pris en considération.

Liu et ses collègues (Liu, Yang, Niu, Ma & Shao, 2009) ont essayé d'analyser les relations entre les différentes sortes de mauvaises odeurs de code et ils ont présenté la nécessité d'organiser un ordre lors de la résolution des mauvaises odeurs. Avec l'analyse, ils ont introduit un framework qui permet au développeur d'effectuer la restructuration automatique de code.

Ganea et ses collègues (Ganea, Verebi & Marinescu, 2017) ont présenté un plug-in Eclipse appelé "InCode", qui évalue en permanence la qualité des systèmes Java. Il aide également les

[1] http://www.eclipse.org/

[2] https://www.jetbrains.com/idea/

développeurs à prendre des décisions de restructuration de code. Les odeurs de code comme la duplication de code et les classes de trop grande taille pourraient être facilement localisées par InCode.

Ekman et Schafer (Ekman, Schäfer & Verbaere, 2008) ont étendu JastAdd en générant un moteur qui effectue automatiquement le refactoring. Ils utilisent des blocs de refactoring simples comme base pour d'autres refactorings tels que le renommage et l'extraction des méthodes.

Chatzigeorgiou et Manakos (Chatzigeorgiou & Manakos, 2010) ont examiné les systèmes Orientés objets pour la détection des odeurs de code et ils ont répondu à plusieurs questions parmi eux nous citons : Si le nombre de problèmes augmente avec le passage des générations de logiciels, si les problèmes disparaissent avec le temps ou uniquement par une intervention humaine ciblée, si de mauvaises odeurs se manifestent au cours de l'évolution d'un module ou subsistent dès le début et si les refactorings visant à éliminer les odeurs sont fréquentes.

Tsantalis et Chatzigeorgiou (Chatzigeorgiou & Manakos, 2010) ont mis en évidence une technique de refactoring en Java en se basant sur le polymorphisme. L'idée proposée a été implémentée sous forme d'un « plugin » Eclipse.

Récemment, de nouvelles approches émergent où des techniques basées sur la recherche ont été utilisées pour automatiser les activités de refactoring. Dans ces approches le refactoring est considéré comme un problème d'optimisation. Ouni et ses collègues (Ouni, Kessentini, Sahraoui & Boukadoum, 2013b)ont proposé une approche d'optimisation multi-objectif pour trouver la meilleure séquence de refactorings en utilisant NSGA-II. L'approche proposée est basée sur deux fonctions objectifs, la qualité et l'effort de développeur pour modifier le code. De plus, dans une autre publication, Ouni (Ouni, Kessentini, Sahraoui & Hamdi, 2012) propose un nouvel outil multi-objectif pour trouver le meilleur compromis entre l'amélioration de la qualité et la cohérence sémantique en utilisant deux heuristiques liées à la similitude du vocabulaire et le couplage structurel. Plus tard (Ouni, Kessentini & Sahraoui, 2013a), ils ont intégré un nouvel objectif, qui vise à maximiser la réutilisation de bonnes techniques de refactoring appliqués à des contextes similaires.

Mkaouer et ses collègues (Mkaouer, Kessentini, Shaout, Koligheu, Bechikh, Deb & Ouni, 2015) ont également proposé une nouvelle approche pour recommander des séquences de restructuration de code basée sur un algorithme génétique multi-objectif NSGA-III. Ils utilisent l'historique de changement de code en tant qu'entrée de leur algorithme pour calculer la similarité d'un candidat de refactoring avec des refactorings antérieurs. Dans leur approche, ils visent à trouver les solutions optimales de modularisation qui améliorent la structure des packages, minimiser le nombre de modifications et préserver la cohérence sémantique.

Oliveira et ses collègues (de Oliveira, Freitas, Bonifácio, Pinto & Lo, 2019) ont présenté une nouvelle approche de recommandation de move method et move field en utilisant l'algorithme génétique multi objectif NSGA-II, qui supprime les dépendances de co-changement et les odeurs de code évolutives. C'est un type particulier de dépendance qui survient lorsque des entités à granularité fine appartenant à différentes classes changent fréquemment ensemble.

Finalement, Boukharata et ses collègues (Boukharata, Ouni, Kessentini, Bouktif & Wang, 2019) ont récemment développé une approche automatique de modularisation des interfaces des services Web, nommée WSIRem. L'approche proposée est basée sur l'optimisation multi-objectifs, en utilisant l'algorithme génétique NSGA-II et elle consiste à trouver la modularisation appropriée d'une interface de service. Leur outil analyse les fichiers WSDL des services web pour extraire les dépendances sémantiques et structurelles entre les opérations d'une interface. Ils ont défini trois fonctions objectifs qui visent à minimiser le couplage, maximiser la cohésion et minimiser les modifications des interfaces. WSIRem a été évalué sur un banc d'essai de 22 services Web d'Amazon et de Yahoo. D'autres travaux ont également exploré la décomposition/extraction des interfaces des services Web (Ouni, Kessentini, Inoue & Cinnéide, 2015 ; Ouni, Salem, Inoue & Soui, 2016 ; Ouni, Wang, Kessentini, Bouktif & Inoue, 2018 ; Wang, Ouni, Kessentini, Maxim & Grosky, 2016 ; Wang, Kessentini & Ouni, 2017) qui s'aligne avec l'extraction de classes mais avec différents technologies. Cependant, ces techniques d'extraction d'interfaces de services Web ne sont pas directement applicables dans le contexte des programmes Java.

Les approches de refactoring automatisés présentent plusieurs avantages par rapport aux deux autres techniques. Parmi ces avantages, nous citons que pour les techniques qui se basent sur les approches méta-heuristiques, la fonction d'évaluation réduit le nombre des odeurs de code détectées dans le système après l'application de certaines opérations de refactoring. Finalement, l'avantage majeur de ces approches est de minimiser l'effort exercé par les développeurs et d'améliorer la qualité de code.

1.3.2 Approches du refactoring "Extract class"

Fokaefs et ses collègues (Fokaefs, Tsantalis, Stroulia & Chatzigeorgiou, 2011) ont utilisé un algorithme de groupement pour extraire les classes. Leur approche "JDeodorant" analyse les dépendances structurelles entre les attributs et les méthodes d'une classe. Ils calculent la distance de Jaccard entre tous les couples d'entités. Ces calculs sont faits afin de composer des groupes d'entités qui peuvent être ensuite extraits en tant que classes séparées. Dans ce cas seules les informations structurelles sont prises en compte pour effectuer l'extraction de la classe. Cette approche se base sur un diagramme en arbre qui représente la sortie d'un algorithme de classification hiérarchique. Les feuilles de l'arbre représentent les entités à regrouper tandis que les nœuds restants représentent les groupes possibles auxquels les entités appartiennent, jusqu'au racine qui représent un groupe contenant toutes les entités. Ils ont évalué l'outil sur le projet open source JHotDraw (version 5.3) en demandant à des professionnels d'examiner le « plugin » et de donner leurs avis.

Joshi et Joshi (Joshi & Joshi, 2009) ont considéré le problème des classes à faible cohésion en tant que problème de partition de graphe. Ils se concentrent sur l'amélioration de la cohésion de classe en examinant les treillis en fonction des dépendances entre les attributs et les méthodes. Le défaut de cette méthode, est que, pour les grands systèmes, les treillis peuvent devenir très complexes et donc il est plus difficile pour le concepteur d'inspecter visuellement le treillis et d'identifier les défauts. De plus, tandis que cette méthode se concentre sur l'amélioration de la cohésion d'une classe, elle néglige de considérer la cohérence conceptuelle des classes extraites

suggérées. Enfin, cette méthode ne garantit pas que les solutions suggérées n'affectent pas le comportement du programme.

De Lucia et ses collègues (De Lucia, Oliveto & Vorraro, 2008) ont proposé une méthodologie qui prend en compte à la fois des critères structurels et conceptuels. Leur approche consiste dans la création d'un graphique pondéré des méthodes de classe basé sur la cohésion structurelle et sémantique. Ce graphique est ensuite divisé en utilisant un algorithme intitulé Max-flow Min-Cut pour produire des classes plus cohérentes. La cohésion sémantique est basée sur les noms des classes et des entités. L'approche proposée a été évaluée dans une étude de cas réalisée sur JHotDraw. Les résultats obtenus ont indiqué que les performances obtenues avec l'approche proposée excellent les résultats obtenus avec des outils ne considérant que des informations structurelles.

Dans un travail similaire, Bavota et ses collègues (Bavota, De Lucia, Marcus & Oliveto, 2010a) ont proposé une technique simple pour identifier les opportunités d'extraction des classes, en fonction des appels entre les méthodes. Ils trouvent des chaînes de méthodes en calculant la fermeture transitive de leurs dépendances. Les chaînes qui sont au-dessus d'une cohésion minimale et sont d'une longueur minimale, sont suggérées comme extractions possibles. L'utilisation des chaînes permet à la technique d'identifier éventuellement plus de deux extractions. Ce travail souffre de certaines limitations trouvées dans ses prédécesseurs, tels que l'exclusion des attributs du processus de partitionnement qui peut conduire à des solutions non-optimales. Ils ont rapporté une évaluation préliminaire de l'approche proposée en proposant un scénario sur deux logiciels libres.

Bavota et ses collègues (Bavota, Oliveto, De Lucia, Antoniol & Gueheneuc, 2010b) proposent aussi une approche de la théorie des jeux pour identifier les opportunités pour extraire une classe. Dans cette approche, les deux classes candidates, dans lesquelles une classe « God class » peut être décomposée, "Rivaliser" les uns contre les autres sur les méthodes de la classe source d'origine. A chaque tour, chaque "joueur" essaie d'obtenir une méthode qui va augmenter sa cohésion et diminuer son couplage basé sur des mesures de similarité structurelle et sémantique.

Une évaluation de cette approche a été réalisée sur deux systèmes open source à savoir ArgoUML version 0.16 et JHotDraw version 6.0b1.

Finalement, Bavota et ses collègues (Bavota, De Lucia, Marcus, Oliveto & Palomba, 2012) ont utilisé SSM (Structural Similarity between Methods), CSM (Conceptual Similarity between Methods) et CDM (Call based dependence between Methods) pour calculer la similarité entre les méthodes. Un graphique est construit avec des nœuds représentant les méthodes de classe et des arêtes entre deux méthodes représentant leur similarité quantitative. Les bords entre les méthodes faiblement similaires sont filtrés à l'aide d'un seuil (couplage min.). Dans ce graphique, les chaînes de méthodes ayant un nombre total de nœuds inférieur à un seuil, minLength, sont appelées chaînes triviales. Les chaînes avec des nœuds supérieurs à minLength sont appelées chaînes non triviales. La similarité entre chaque chaîne triviale et non-triviale est évaluée en faisant la moyenne de la similarité (basée sur SSM, CSM et CDM) pour chaque paire de méthodes et utilisée pour fusionner des chaînes triviales avec la chaîne non-triviale la plus similaire. Ensuite, chaque chaîne correspond à une classe distincte où les méthodes de la chaîne finissent en tant que méthodes de la classe. En cas de conflit entre les classes sur une variable, celle-ci est attribuée à la classe qui l'utilise dans plusieurs méthodes. Si deux ou plusieurs classes utilisent la variable dans le même nombre de méthodes, il est attribué à la plus petite classe (ayant moins de méthodes).

- **Limites des approches existantes :**

La conclusion à tirer des travaux de restructuration de classe existants est que certains travaux ont été basées principalement sur les dépendances structurelles seulement tels que Fokaefs et ses collègues (Fokaefs *et al.*, 2011), Joshi et Joshi (Joshi & Joshi, 2009), Bavota et ses collègues (Bavota *et al.*, 2010a). En outre, ces approches et outils ne reposent que sur des informations statiques qui ne suffisent pas toujours pour comprendre et préserver la cohérence sémantique du code source lors de la recommandation de la restructuration de la classe. Autres approches ont été proposées comme De Lucia et ses collègues (De Lucia *et al.*, 2008), Bavota et ses

collègues (Bavota *et al.*, 2010a) et (Bavota *et al.*, 2012) qui ont considéré la similarité sémantique entre les entités de la classe. D'autres aspects pourraient contribuer de manière significative à l'efficacité des systèmes de recommandation de restructuration des classes, tels que l'utilisation de l'historique des changements de code. Comme de nombreuses combinaisons des membres sont possibles, le processus de génération de recommandations est, par nature, un problème d'optimisation combinatoire. Une recherche déterministe n'est pas pratique dans de tels cas et l'utilisation de la recherche heuristique est justifiée. À cette fin, une recherche heuristique est nécessaire pour explorer les solutions possibles de recommandation de restructuration des classes de trop grande taille.

1.4 Conclusion

Ce chapitre a été consacré pour définir les concepts de base de ce mémoire et aussi pour mettre l'accent sur les travaux connexes. À la fin de ce chapitre, nous avons étudié les limites des approches précédentes et nous avons aussi justifié l'apport de notre solution que nous allons présenter en détail dans le chapitre suivant.

CHAPITRE 2

PRÉSENTATION DE L'APPROCHE PROPOSÉE

2.1 Introduction

Dans le cadre de ce travail, notre objectif consiste à proposer et à concevoir une approche automatique de recommandation des solutions de restructuration des classes de trop grande tailles. Tout d'abord, nous commençons ce chapitre par présenter un exemple de motivation pour illustrer notre approche. Nous décrivons ensuite, les différentes étapes de notre approche.

2.2 Exemple de motivation

Pour illustrer le besoin de notre approche, nous présentons dans cette section un exemple réel. Prenant le projet Jfreechart v1.5.0 [1], un projet open source implémenté en java. Nous observons que la classe "`StandardChartTheme`" est considérée par l'outil PMD [2] (un outil qui analyse le code source Java et détecte les odeurs de code) comme une classe de trop grande taille puisqu'elle contient 92 méthodes.

Comme premier essai, nous avons appliqué "JDeodorant" [3], un plug-in Eclipse très utilisé pour détecter les odeurs de code Java et recommander des opérations de refactoring appropriées pour les résoudre. Cet outil a recommandé la solution présentée par la Figure 2.1. En ré-appliquant une deuxième fois l'outil de détection des odeurs de code PMD sur le projet, la classe "`StandardChartTheme`" apparaît toujours comme une classe de trop grande taille. Ceci implique que JDeodorant n'a pas pu résoudre le problème pour cet exemple. En effet, JDeodorant se base que sur une seule heuristique à savoir les dépendances structurelles entre les entités de la classe. Le résultat de l'application de JDeodorant sur cet exemple a montré que cette heuristique est insuffisante pour restructurer la classe considérée.

[1] https://github.com/jfree/jfreechart

[2] https://pmd.github.io/

[3] https://github.com/tsantalis/JDeodorant

Figure 2.1 Restructuration de la classe StandardChartTheme avec JDeodorant

Comme qu'illustré dans la Figure 2.1, cette approche a extrait une seule classe où les deux méthodes accèdent au même attribut "`drawingSupplier`". Cette extraction est correcte mais d'un autre côté, elle a laissé les autres entités dans la même classe. Parmi ces entités, nous pourons identifier des méthodes qui sont sémantiquement similaires tels que "`getExtraLargeFont()`" et "`getLargeFont()`" et qu'ils sont dissimilaires à un autre ensemble des membres de la classe tels que "`getXYBarPainter()`" et "`getThermometerPaint()`". Aussi des entités qui ont été modifiées ensemble par le développeur en même temps et qu'elles ont été laissées dans la même classe.

En analysant les dépendances structurelles entre les membres de la classe. Nous constatons que, malgré que JDeodorant se base sur ces dépendances et extrait les entités de la classe qui sont liées ensemble structurellement, cette approche a laissé un grand ensemble des membres qui sont structurellement dépendants entre eux, mais indépendant avec le reste des membres dans la même classe. Cette limite confirme que le problème de restructuration d'une classe de trop grande taille ne peut pas être résolu correctement avec une approche déterministe.

Alors, nous observons que le problème d'une classe avec plusieurs responsabilités est toujours présent. En effet, dans la programmation orientée objet une classe doit respecter le principe de la responsabilité unique (PRU) tel que décrit par Robert Martin (Martin, 2002). Martin montre que c'est très important de séparer les responsabilités en différentes classes. Pour séparer ces méthodes, il faut prendre en compte d'autres aspets de dépendances.

Pour cette raison, nous avons pensé à mettre en place notre approche qui permet de résoudre le problème d'extraction des classes de trop grande taille en utilisant trois heuristiques : la dépendance structurelle, la dépendance sémantique et le co-changement entre les méthodes de la classe.

Si nous appliquons l'idée d'une approche métaheuristique en utilisant trois heuristiques : la dépendance structurelle, la dépendance sémantique et le co-changement. Nous allons obtenir une recommandation de restructuration différente de celle dans la Figure 2.1. La solution est présentée par la Figure 2.2. En analysant le résultat, nous observons que les méthodes extraites sont reliées ensemble d'une façon structurelle et sémantique. Nous constatons aussi qu'elles ont été modifiées ensembles plusieurs fois par les développeurs.

D'un point de vue dépendance structurelle, nous constatons que les méthodes extraites ensembles ont une forte dépendance. Si nous observons en détail la classe source, nous voyons qu'il existe un ensemble des méthodes qui accèdent à des attributs spécifiques. En plus que les méthodes de lecture et de modification de chaque attribut (get et set), il existe quelques méthodes qui ont accès à un ensemble des attributs tandis que d'autres méthodes n'ont aucune relation structurelle avec ces derniers. Par exemple la méthode "`createDarknessTheme()`" a

Figure 2.2 Restructuration de la classe StandardChartTheme avec notre outil

accès aux attributs : “`titlePaint`”, “`subtitlePaint`”, “`chartBackgroundPaint`”, “`legendBackgroundPaint`”, “`legendItemPaintet`” “`plotBackgroundPaint`”. Cette méthode a été extraite avec les méthodes accesseurs et mutateurs des attributs cités avant.

Un autre point important à signaler est que la plupart des méthodes extraites sont sémantiquement similaires. En effet, la similarité sémantique entre la plupart de ces méthodes pour chaque classe est très importante. Prenant l’exemple de la troisième classe extraite “`StandardChartThemeProduct3`”, entre la méthode “`getRegularFont()`“ et la méthode “`getSmallFont()`”, la similarité sémantique est égale à 0.79. La similarité entre

la première méthode et la méthode "`setRegularFont()`", est maximale et égale à 1, en éliminant toujours les mots réservés (void, string, int, ...) et les accesseurs (get et set).

Finalement, quelques entités de la classe n'ont aucune liaison structurelle ou sémantique entre eux et avec ça ils ont été extraites. Après une analyse profonde des résultats, nous avons trouvé que ces membres ont été modifiés ensemble par le développeur simultanément et plusieurs fois.

Par exemple, en analysant l'historique de changements sur GitHub [4], la classe "`StandardChart−Theme`" a été changée 42 fois. La méthode "`apply(Jfreechart)`" a été modifiée 21 fois par les développeurs. Parmi ces modifications, nous trouvons que cette méthode a été changée 10 fois avec la méthode "`createDarknessTheme()`". Sémantiquement, ces deux méthodes ne se rassemblent pas et aucune dépendance structurelle n'existe entre eux. En plus, la similarité sémantique maximale de la méthode "`apply(Jfreechart)`" avec les restes des méthodes de la classe, est de l'ordre de 0.22, qui n'est pas significative.

Finalement, nous avons calculé le couplage et la cohésion en utilisant CK [5] et jPeek [6], qui sont deux outils open source qui calculent des métriques de qualité dans les projets Java au moyen d'une analyse statique, avant et après le refactoring. Nous avons trouvé les résultats présentés dans le Tableau 2.1.

Tableau 2.1 Couplage et cohésion de la classe avant et après refactoring

Approche	**Couplage : CBO**	**Cohésion**	
		LCOM	**CAMC**
Avant refactoring	52	3048	0.03
JDeodorant	53	1523.5	0.22
Notre approche	56	462	0.36

Le couplage de la partition (CBO :Coupling Between Objects) après l'extraction de la classe a augmenté par 1 pour le cas de JDeodorant et par 4 pour le cas de notre approche. Ceci a du sens

[4] https://github.com/

[5] https://github.com/mauricioaniche/ck

[6] https://github.com/yegor256/jpeek

parce que tant que la classe a été devisée et 3 autres classes ont apparu, le nombre de liaisons va augmenter.

Pour évaluer les résultats par rapport à la cohésion, nous avons utilisé deux métriques de cohésion. LCOM (Lack of Cohesion Of Methods), plus la valeur de LCOM est grande plus la classe est moins cohésive. La deuxième métrique est CAMC (Cohesion Among Methods Of a Class), pour cette métrique plus la valeur de CAMC est proche de 1 plus la cohésion est bonne. Avant le refactoring la cohésion a été faible et égale à 0.03. Avec l'outil JDeodorant, nous constatons une amélioration de la cohésion de la partition (0.22, en moyenne). Avec notre outil, le résultat de CAMC montre une amélioration importante de la cohésion de la partition qui a passé en moyenne de 0.03 à 0.36.

Sur la base de ces observations, nous déduisons que, si nous tenons compte que des dépendances structurelles entre les entités de la classe, nous allons avoir des recommandations qui ne sont pas nécessairement optimales pour l'extraction de classes. Donc il faut prendre en considération d'autres heuristiques pour résoudre ce problème, tel que la similarité sémantique et le co-changement. Nous allons évaluer le choix et la valeur ajoutée de chaque métrique de qualité dans le chapitre suivant.

2.3 Description de l'approche

Dans cette section nous expliquons notre approche, BlobBreak. Le but de BlobBreak est de chercher les meilleures recommandations de restructuration d'une classe de trop grande taille. La Figure 2.3 illustre la structure générale de notre approche. BlobBreak, prends en entrée le code source (Java) et le nom d'une classe de trop grande taille spécifiée par le développeur. Il utilise l'outil d'analyse statique de code source JDT (Java Development Tools) d'Eclipse pour analyser les relations entre les classes, les méthodes, les attributs, les variables, etc. L'outil génère un graphe de dépendance dirigé G = (V, E) pour l'ensemble du programme où les sommets V représentent les méthodes et les attributs, et les arêtes E représentent les dépendances (appels de méthodes et accès aux attributs) entre eux.

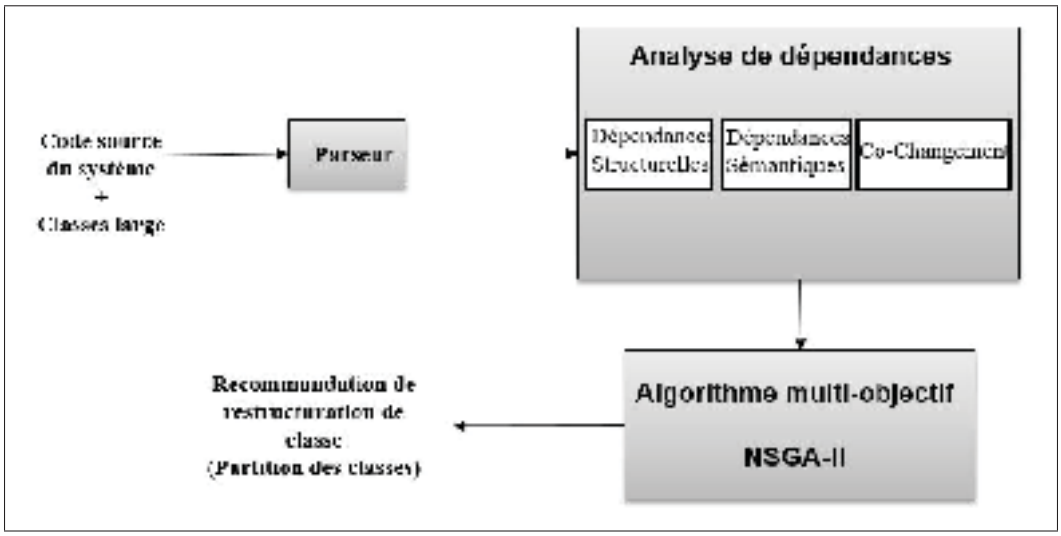

Figure 2.3 Description de l'approche proposée : BlobBreak

Après l'analyse de code, il génère un ensemble de mesures sémantiques, structurelles, ainsi que des mesures liées à l'historique des changements de code appliqués aux versions précédentes du système. Ces mesures seront par la suite, l'entrée de l'algorithme génétique multi-objectif. BlobBreak génère en sortie une liste de toutes les recommandations qui améliorent la qualité de la classe. Un exemple simplifié de restructuration d'une classe de trop grande taille est présenté par la Figure 2.4 qui contient une classe de trop grande taille (C) avec 3 attributs et 4 méthodes, qui a été restructurée sur 2 classes (C-1 et C-2). Dans ce qui suit, nous décrivons les trois heuristiques utilisés dans notre approche, en se basant sur cet exemple.

2.3.1 Dépendance structurelle

Notre approche s'appuie sur l'ensemble des dépendances statiques établies par une méthode m pour calculer sa similarité avec tous les autres membres d'une classe C, qu'on appelle StructDep(mi). StructDep(mi) est un ensemble de dépendances établi par la méthode mi dans la classes C. Plus précisément, nous considérons les dépendances suivantes :

- Appels de méthode : m2 appelle m3, m3 est ajoutée à StructDep(m2).

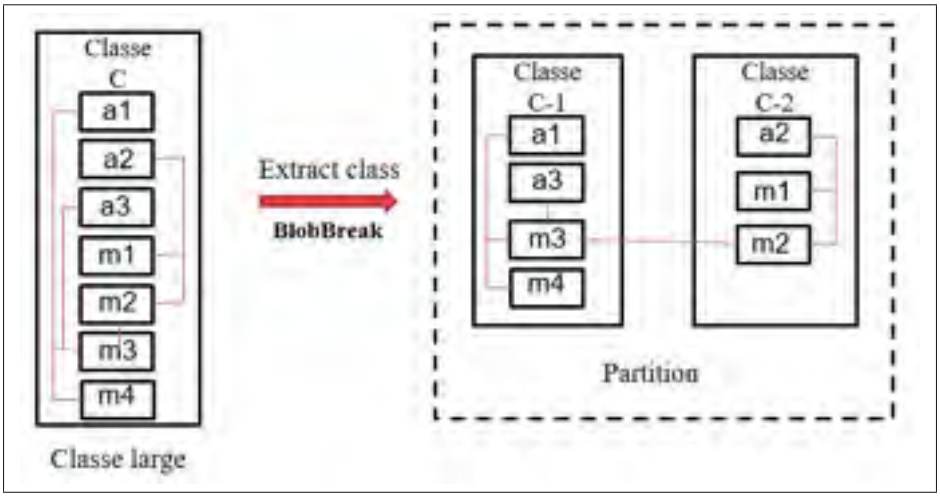

Figure 2.4 Exemple de restructuration de classe

- Accès aux champs : m2 lit ou écrit dans un champ d'un attribut a2, dans ce cas a2 est ajouté à StructDep(m2).

Lors de la construction des ensembles de dépendances, nous ignorons les types primitifs et les annotations de java.lang et java.util (comme String, HashTable, Objet, et SupressWarnings). Ces types sont communs à la plupart des classes et pour cette raison, ils ne sont pas pertinents lorsqu'on parle de dépendances établies par les méthodes.

Nous définissons les arêtes internes et intermédiaires (externes) dans les nouvelles classes après la restructuration. Les arêtes internes (Intra) sont celles pour lesquelles la source et la cible de l'arête sont des éléments appartenant à la même classe (par exemple, l'arête comprise entre m3 et m4 dans l'exemple illustré à la Figure 2.4). Les arêtes intermédiaires (Inter) sont celles pour lesquelles la source et la cible appartiennent à des classes distinctes (par exemple, l'arête entre m3 et m2 dans l'exemple illustré à la Figure 2.4). Le nombre d'accès interne (Intra) est stocké à chaque fois dans une liste IntraList. Aussi, une liste InterList est mise en œuvre pour enregistrer le nombre d'accès externe (Inter).

Les dépendances internes et intermédiaires de la classe C sont calculées de cette façon :

$$IntraStuct(C) = \frac{\sum_{i=1}^{T} IntraList(i)}{max * T} \tag{2.1}$$

Avec max est la valeur maximale de la liste IntraList et T est sa taille.

$$InterStruct(P) = \frac{\sum_{j=1}^{T} InterList(j)}{max * T} \tag{2.2}$$

Avec max est la valeur maximale de la liste InterList et T est sa taille.

2.3.2 Dépendance sémantique

En plus des dépendances structurelles, BlobBreak utilise une analyse sémantique en extrayant tous les identifiants de code, y compris les noms des méthodes et attributs, le type de retour et les paramètres pour chaque méthode, ainsi que les variables déclarées dans les méthodes.

Cette heuristique vise à garantir que les classes extraites ont des responsabilités distinctes. Cette heuristique est basée sur le principe de responsabilité unique. Ce principe énonce que chaque classe doit avoir exactement une responsabilité, être liée à une seule exigence fonctionnelle. Notre intuition ici, est que les classes avec de différentes responsabilités peuvent avoir un contenu sémantique différent. Alors, en plus de dépendances structurelles, nous considérons également la similarité sémantique entre les entités de la classe et les classes extraites pour guider le processus de recherche. Cette heuristique est basée sur l'hypothèse que les responsabilités d'une classe sont figurées dans le vocabulaire et la terminologie utilisés pour implémenter cette classe.

Nous définissons la similarité sémantique entre les entités de la classe **SS(mi,mj)** comme suit : deux membres sont considérés comme sémantiquement liées si leurs sémantiques sont similaires, elles effectuent des actions conceptuellement similaires. Pour mesurer la similarité sémantique, un modèle tf-idf (Salton & Buckley, 1988) est utilisé pour représenter chaque méthode en tant que vecteur couvrant un espace défini par le vocabulaire extrait du contenu de la méthode. Le vocabulaire d'une méthode est extrait du nom de la méthode, de paramètres, de variables locales,

au sein d'une méthode et de son type de retour. Le vocabulaire d'un attribut est le nom de l'attribut ainsi que son type. Après le traitement préalable en utilisant la division de la casse, suivie du filtrage des mots tels que les mots réservés (string, int, statique, private, etc). La similarité sémantique entre deux membres d'une classe m1 et m2 est ensuite calculée comme étant le cosinus de l'angle entre les vecteurs correspondants, comme suit :

$$SS(mi, mj) = \frac{\vec{mi}.\vec{mj}}{\left\|\vec{mi}\right\| . \left\|\vec{mj}\right\|} \tag{2.3}$$

Avec $\vec{mi}$ et $\vec{mj}$ sont les vecteurs qui correspondent aux membres mi et mj, respectivement. Les $\left\|\vec{mi}\right\|$ et $\left\|\vec{mj}\right\|$ représentent la distance euclidienne entre les vecteurs $\vec{mi}$ et $\vec{mj}$.

Ensuite, on calcule de la même façon la similarité sémantique, mais cette fois entre les classes extraites **SS(Ci,Cj)**. Le vocabulaire d'une classe est extrait des identifiants des attributs, méthodes, variables, paramètres, commentaires, au sein d'une classe. La similarité sémantique entre deux classes c1 et c2 est ensuite calculée par le cosinus de l'angle entre les vecteurs correspondants, comme suit :

$$SS(Ci, Cj) = \frac{\vec{Ci}.\vec{Cj}}{\left\|\vec{Ci}\right\| . \left\|\vec{Cj}\right\|} \tag{2.4}$$

Avec $\vec{ci}$ et $\vec{cj}$ sont les vecteurs qui correspondent aux classes ci et cj, respectivement. Les $\left\|\vec{ci}\right\|$ et $\left\|\vec{cj}\right\|$ représentent la distance euclidienne entre les vecteurs $\vec{ci}$ et $\vec{cj}$.

Après la similarité sémantique de toute la classe est calculée par l'équation 2.5 :

$$MSS(C) = \frac{\sum_{\forall(mi,mj)\in C} SS(mi, mj)}{\frac{(|C|*(|C|-1))}{2}} \tag{2.5}$$

Avec |C| est le nombre des membres dans la classe (C).

Et la similarité sémantique de la partition des classes extraites est sous cette forme :

$$CSS(P) = \frac{\sum_{\forall(Ci,Cj)\in P} SS(Ci, Cj)}{\frac{(|P|*(|P|-1))}{2}} \tag{2.6}$$

Avec |P| est le nombre des classes dans la partition (P).

2.3.3 Changement de code

Le but de cette heuristique est de regrouper des méthodes qui changent fréquemment ensemble. Nous partons de l'hypothèse que les méthodes qui sont fréquemment modifiées ensemble sont susceptibles d'avoir une implémentation liée, ce qui appartient donc à la même fonctionnalité.

Notre hypothèse est également basée sur le principe de responsabilité unique (Martin, 2002), qui suggère que chaque classe doit avoir une seule responsabilité et une seule raison pour être changée. Par conséquent, nous exploitons l'historique de développement d'un projet logiciel disponible dans leur système de contrôle de version SCV. Les SCVs tels que Git aident à maintenir l'évolution des artéfacts de code source de manière fiable. Un utilisateur d'un SCV soumet des ensembles de modifications (impliquant un ou plusieurs artéfacts) sous la forme d'un "commit". En conséquence, l'historique des modifications soumises à un système de contrôle de version peut être décrit comme une séquence de "commits", où chaque "commit" contient un sous-ensemble d'artéfacts qui ont été modifiés ensemble. Pour extraire les "commits" de code nous avons utilisé l'outil repodriller [7] (Bird, Rigby, Barr, Hamilton, German & Devanbu, 2009). Cet outil se base sur l'API de GitHub et nous donne tous les changements qui ont été appliqués par les développeurs dès la date de premier "commit" de projet Java.

Nous analysons ces modifications pour la recherche des changements que notre classe a subis. Ensuite, nous sauvegardons toutes les versions de code source. À partir de ces versions, nous extrayons les signatures des méthodes qui ont été modifiées ensemble en même temps.

De manière similaire à la première heuristique, nous calculons le co-changement entre les membres de la classe. En particulier, nous définissons le score de changement entre deux méthodes comme le nombre de "commits" dans lesquels ces deux méthodes ont changé ensemble, puis nous calculons de la même manière que la mesure de dépendances structurelles, un score IntraCH entre les méthodes de la même classe est stocké à chaque fois dans une liste

[7] https://github.com/mauricioaniche/repodriller

IntraCHList et un score InterCH entre les méthodes des classes différentes est enregistré dans la liste InterCHList.

$$IntraCH(C) = \frac{\sum_{i=1}^{T} IntraCHList(i)}{max * T} \tag{2.7}$$

Avec max est la valeur maximale de la liste IntraCHList et T est sa taille.

$$InterCH(P) = \frac{\sum_{j=1}^{T} InterCHList(j)}{max * T} \tag{2.8}$$

Avec max est la valeur maximale de la liste InterCHList et T est sa taille.

Dans ce qui suit, nous allons expliquer comment l'algorithme génétique est adapté au problème d'extraction de classe.

2.4 Adaptation de l'algorithme génétique NSGA-II

Pour adapter NSGA-II à un problème spécifique, les éléments suivants doivent être définis : la représentation de la solution, la génération de la population initiale, les fonctions objectifs pour évaluer les solutions candidates et les opérateurs génétiques pour la génération d'une nouvelle population et les contraintes. Nous décrivons ci-après ces éléments.

2.4.1 Présentation de la solution

Une solution potentielle au problème est un partitionnement de classe, c'est-à-dire un ensemble de classes(tel que l'exemple de la Figure 2.4), chacune exposant un ensemble de membres cohérents (méthodes et attributs). Une solution valide affecte chaque membre à une seule classe et ne contient aucune classe vide. En particulier, nous adoptons le codage entier dans lequel une solution candidate est représentée par un tableau entier de n positions, où n est le nombre de membres exposés dans une classe. Chaque poste correspond à un membre du groupe spécifique. Par exemple, dans l'exemple de la Figure 2.5, où les membres a1, m2 et m5 appartiennent à la même classe C1 et les membres a2, m1et m6 appartiennent à la classe C2.Enfin les membres a3,

m3 et m7 appartiennent à la classe C3. La population initiale est complètement aléatoire où un

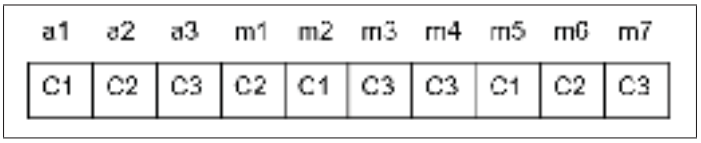

Figure 2.5 Présentation de la solution

nombre maximum de classes m est fixé, puis chaque membre de la classe d'origine est assigné aléatoirement à une classe unique.

2.4.2 Fonctions objectifs

Pour évaluer la qualité d'une partition (P) candidate d'une classe source (C), nous avons utilisé plus qu'une fonction objectif conçue en se basant sur les heuristiques décrites dans la section précédente. En particulier, l'adéquation d'une solution candidate est évaluée en fonction de deux fonctions objectifs suivantes :

1. Minimiser les dépendances externes entre les classes extraites :

$$F1 = \frac{InterStruct(P) + CSS(P) + InterCH(P)}{3} \tag{2.9}$$

2. Maximiser la cohésion interne des classes extraites :

$$F2 = \frac{IntraStruct(C) + MSS(C) + IntraCH(C)}{3} \tag{2.10}$$

2.4.3 Opérateurs génétiques

A) Opérateur de sélection

Pour sélectionner les individus candidats, nous adoptons la sélection par tournoi. Cette méthode consiste à choisir aléatoirement un sous-ensemble d'individus dans la population, puis à sélectionner le meilleur.

B) L'opérateur de croisement

Pour le croisement, nous utilisons l'opérateur de croisement à un point comme montré dans la Figure 2.6. Nous commençons par sélectionner et fractionner au hasard deux solutions parentales. Ensuite, cet opérateur crée deux solutions enfants en mettant : Dans l'exemple, i représente la

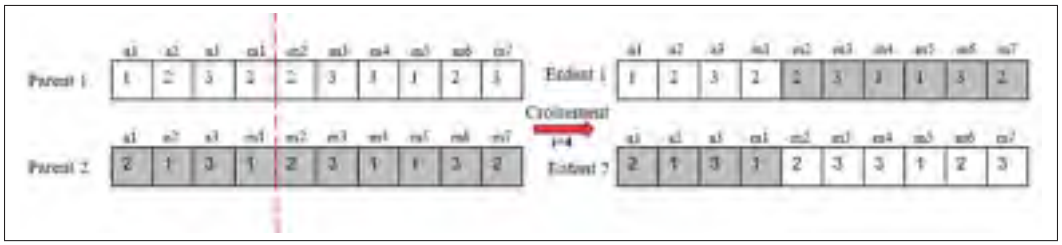

Figure 2.6 Croisement

position de croisement choisie au niveau des parents.

C) L'opérateur de mutation

l'opérateur de mutation est utilisé pour introduire de légères modifications aléatoires dans les solutions candidates. Cet opérateur guide l'algorithme dans des zones de l'espace de recherche qui ne seraient pas accessibles par seulement la recombinaison.

Dans notre approche, nous utilisons un opérateur de mutation qui sélectionne au hasard une ou plusieurs positions de leur tableau d'entiers et les remplace par d'autres au hasard, comme illustré à la Figure 2.7. Pour être valides, les opérateurs de croisement et de mutation doivent s'assurer que chaque membre de la classe est affecté à une classe unique et que la population respecte les contraintes citées dans la section suivante .

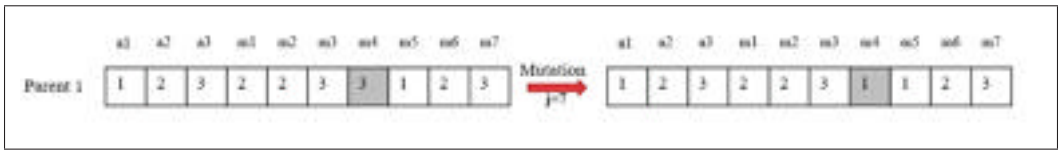

Figure 2.7 Mutation

2.4.4 Contraintes

Nous avons défini le paramètre minMembers comme le nombre minimum de membres par classe extraite. Nous avons fixé ce nombre à 3 pour éviter d'avoir des classes trop petites. Nous avons également défini le paramètre AttperClass comme le rapport entre le nombre d'attributs et le nombre total de membres de la classe. Ce paramètre vise à contrôler le nombre d'attributs dans les classes pour éviter d'avoir des classes avec uniquement des attributs.

2.5 Présentation du « plugin »

Nous avons implémenté notre solution sous la forme d'un « plugin » Eclipse présenté par la Figure 2.8. Pour la mise en oeuvre de l'algorithme génétique multi-objectif NSGA-II, nous avons utilisé la bibliothèque MOEA [8] de David Hadka (Hadka & Reed, 2013). Cette bibliothèque permet de développer et d'expérimenter des algorithmes évolutifs multi-objectifs. Notre « plugin » prend comme entrée le code source de système ainsi que le nom de la classe de trop grande taille spécifiée par l'utilisateur. Il génère en sortie plusieurs solutions de recommandation optimales. Nous sauvegardons ces solutions dans un fichier texte accessible par l'utilisateur (Figure 2.9). Nous affichons aussi dans la vue "Extract Class" une seule solution choisie par l'outil. Aussi, l'utilisateur a la possibilité de revenir au fichier texte et de choisir une autre solution qui convient à ses besoins.

[8] http://moeaframework.org/

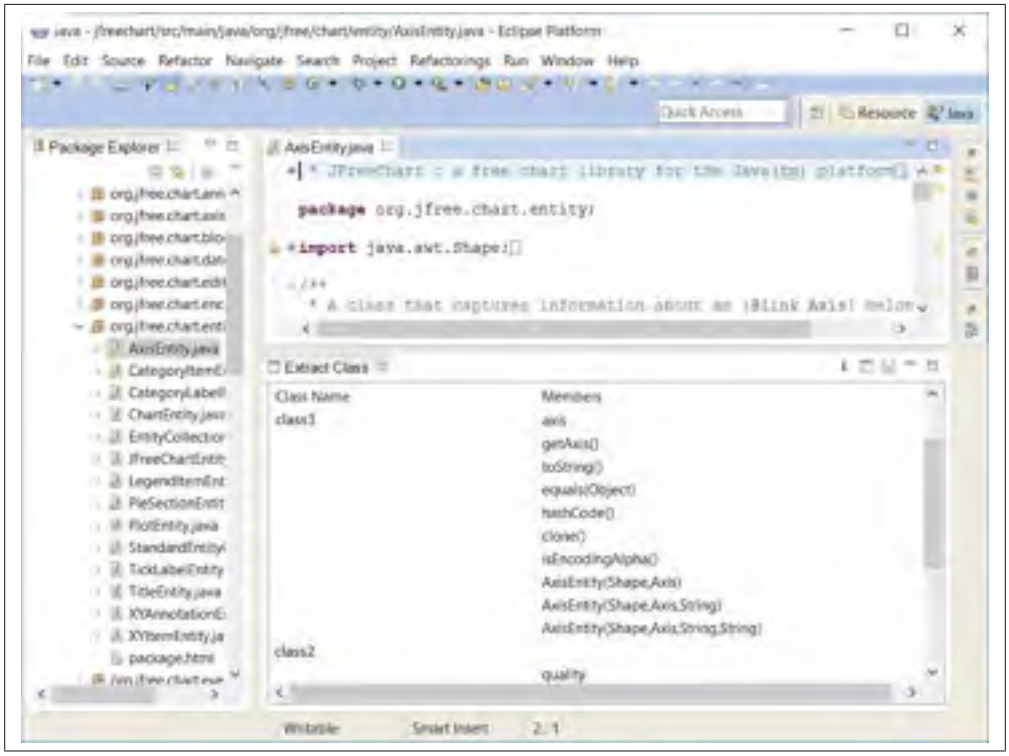

Figure 2.8 Présentation du « plugin »

Figure 2.9 Fichier texte contenant
les solutions optimales

2.6 Conclusion

Dans ce chapitre, nous avons présenté l'approche proposée, nommée BlobBreak pour le refactoring des classes de trop grande taille. BlobBreak consiste à adapter NSGA-II pour générer les meilleures solutions de restructuration des classes afin d'améliorer leurs qualités. L'objectif du prochain chapitre est la présentation des détails de nos expérimentations ainsi que les résultats obtenus.

CHAPITRE 3

ÉVALUATION

3.1 Introduction

Afin d'évaluer la faisabilité et l'efficacité de notre approche, BlobBreak, à générer des recommandations du refactoring "Extract Class", nous avons mené un ensemble d'expérimentations sur différentes versions de logiciels Java open source. Dans ce chapitre, nous commençons par poser nos questions de recherche. Ensuite, nous décrivons la conception de nos expérimentations et discutons les résultats obtenus.

3.2 Questions de recherche

Nous évaluons les performances de notre approche en étudiant si elle pourrait générer de bonnes recommandations du refactoring "Extract Class" permettant de corriger l'odeur de code de classe de trop grande taille. Notre étude vise à répondre aux quatre questions de recherche (QR). Nous expliquons également comment nos expérimentations sont conçues pour répondre à ces questions. Les quatre questions de recherche sont les suivants :

- **QR1 :** (Comparaison entre les algorithmes multi objectifs) Comment l'algorithme NSGA-II se comporte-il, en le comparant avec deux autres algorithmes génétiques multi-objectifs ?
- **QR2 :** (Précision, rappel et taux d'échec) Dans quelle mesure notre approche, BlobBreak, peut-elle corriger le problème de classe de trop grande taille correctement ?
- **QR3 :** (Cohésion et couplage) Dans quelle mesure notre approche, BlobBreak, peut-elle améliorer la qualité des classes en comparaison avec JDeodorant ?
- **QR4 :** (Impact des heuristiques) Quelle est l'impact de chaque heuristique utilisée, structurelle, sémantique et co-changement, dans notre approche ?

3.3 Systèmes étudiés

Pour évaluer notre approche nous avons utilisé cinq projets Java du domaine du logiciel libre de moyenne et grande taille : Xerces2-J [1], JFreeChart [2], GanttProject [3], AntAppache [4] et Rhino [5]. Xerces2-J est une famille de progiciels d'analyse XML. JFreeChart est une bibliothèque pour générer des graphiques. GanttProject est un outil multiplateforme pour la planification de projets. AntAppache est une bibliothèque Java et un outil de ligne de commande dont la mission est de piloter les processus décrits dans les fichiers de compilation en tant que cibles et points d'extension dépendants les uns des autres. Finalement, Rhino est une implémentation open-source de JavaScript entièrement écrit en Java et c'est un interpréteur et compilateur développé pour le navigateur Mozilla / Firefox. Nous avons sélectionné ces systèmes pour notre étude expérimentale car ils ont été fréquemment étudiés dans les travaux précédents. En outre, ils impliquent différents domaines d'applications et ont des tailles différentes allant de 28 à 91174 KLOC avec un grand nombre d'odeurs de code de type classe de classe de trop grande taille. Le Tableau 3.1 fournit des statistiques descriptives sur ces cinq systèmes.

Tableau 3.1 Liste des systèmes

Système	Version	Classes	KLOC	Classes de trop grande taille	Commits
Ant Apache	1.8.2	1191	255	110	14582
Xerces	2.7.0	991	240	114	5510
Jfreechart	1.5.0	924	91174	48	3670
GanttProject	2.1.1	273	28	24	2893
Rhino	1.7.6	305	42	58	3608

[1] https://github.com/apache/xerces2-j

[2] https://github.com/jfree/jfreechart

[3] https://github.com/bardsoftware/ganttproject

[4] https://github.com/apache/ant

[5] https://github.com/mozilla/rhino

3.4 Méthodes d'analyse

Pour répondre à **QR1**, nous avons effectué une comparaison entre 3 algorithmes génétiques multi objectifs NSGA-II, NSGA-III et SPEA2 en utilisant l'outil de MOEA Framework qui fournit un ensemble des métriques de performance dans l'analyse. Cela comprend l'hyper volume, la distance générationnelle et l'espacement.

- L'hypervolume calcule le volume de l'espace dominé par l'ensemble des solutions ;
- La distance générationnelle mesure la distance moyenne entre chaque solution de front Pareto et la solution la plus proche de l'ensemble de référence ;
- L'espacement mesure l'uniformité de l'espacement entre les solutions.

Nous avons réalisé 50 exécutions de chaque algorithme et les résultats obtenus sont analysés statistiquement en utilisant les tests de Kruskal-Wallis (Kruskal et al., 1952) et Mann-Whitney U (Mann & Whitney, 1947) avec un niveau de confiance de 95% (α = 5%). Cette comparaison entre les algorithmes nous permet de valider le choix de NSGA-II.

Pour répondre à **QR2**, nous avons conçu une expérience en combinant un ensemble de classes, puis évalué si BlobBreak et JDeodorant réussissent à extraire les classes originales. Nous avons choisi, aléatoirement, parmi les classes de chaque système, un ensemble de classes (voir Annexe I) à condition qu'elles ne soient pas des classes de trop grande taille. Ensuite, nous avons effectué cinq combinaisons de 2, 3 et 4 classes de façon que ces classes deviennent des classes de trop grande taille après la fusion. Après, nous avons utilisé JDeodorant pour identifier les recommandations d'extraction de classe et nous avons appliqué ces refactorings. Ensuite, nous avons appliqué les recommandations issues de BlobBreak. Finalement, nous avons calculé la précision (P) et le rappel (R) de chaque approche de cette façon :

$$P = \frac{\sum_{i=1}^{m} \frac{\textit{Nombre des membres correctement classés}}{\textit{Nombre des membres par classe proposée}}}{n} \tag{3.1}$$

$$R = \frac{\sum_{i=1}^{m} \frac{\textit{Nombre des membres correctement classés}}{\textit{Nombre des membres par classe originale}}}{n} \tag{3.2}$$

Avec

m= Nombre de classes originales ;

n= max (Nombre de classes originales, Nombre de classes extraites)

Nous avons rapporté aussi le ratio de cas ayant échoué sur le nombre total de classes examinées, calculé par l'Équation (3.3). Un échec étant défini comme un cas où une approche n'extrait aucune classe c'est-à-dire quelle laisse la classe de trop grande taille telle qu'elle est et ne propose aucune recommandation de restructuration de la classe, que ce soit pour JDeodorant ou pour notre approche Pour cette raison, il est évident que ni la précision ni le rappel ne peuvent être calculés.

$$\text{Taux d'échecs} = \frac{\textit{Nombre d'échecs}}{\textit{Nombre total des classes examinées}} \tag{3.3}$$

Pour mieux évaluer notre approche, nous examinons aussi l'amélioration de la qualité en étudiant dans la question suivante la qualité des systèmes avant et après l'application de refactoring Extract Class par BlobBreak ainsi que par JDeodorant. Pour répondre à **QR3**, nous avons évalué la qualité de la classe, en calculant la cohésion et le couplage de la classe avant et après l'application de refactoring recommandé. Nous avons comparé aussi les résultats avec JDeodorant. Pour cela, nous avons utilisé l'outil PMD de détection des odeurs de code. Nous avons choisi aléatoirement 10 classes de trop grande taille (voir Annexe II) pour chaque projet de Tableau 3.1. Par la suite, nous avons calculé les métriques de qualité CBO, LCOM et CAMC sur les systèmes originaux et après l'application de refactoring pour BlobBreak et JDeodorant.

- CBO est le nombre de classes couplées à la classe de trop grande taille et les classes extraites ;
- LCOM et CAMC sont deux indicateurs sur la cohésion de chaque classe.

Pour répondre à **QR4**, nous avons choisi, aléatoirement, 10 classes qui ne sont pas des classes de trop grande taille. Puis nous avons combiné chaque 2 classes ensemble pour avoir à la fin 5 combinaisons de classes. Ensuite nous avons exécuté BlobBreak avec plusieurs combinaisons d'heuristiques. (1) sémantique+ structurelle+ cochange, (2) sémantique+ structurelle, (3) sémantique+ cochange, (4) structurelle+ cochange. Les résultats de ces différentes combinaisons

ont été comparés en termes de précision et rappel. Cette expérimentation permet de valider l'utilité et la contribution de chaque heuristique considérée.

3.5 Résultats

Dans cette section nous allons présenter les résultats des quatre questions de recherche.

3.5.1 Résultats de QR1

Nous avons comparé les résultats de notre approche en utilisant trois algorithmes génétiques multiobjectifs NSGA-II, NSGA-III et SPEA2. L'outil de MOEA Framework founit une analyse statistique et affiche les valeurs minimale, médiane et maximale des indicateurs de performance. Ces indicateurs sont l'hyper volume (HV), la distance générationnelle (DG) et l'espacement (SP). Pour le paramétrage des algorithmes, nous avons fixé le taux de croisement à 0.8 et le taux de mutation à 0.1. Dans le Tableau 3.2, nous comparons les algorithmes par rapport aux valeurs de médian des métriques pour les projets JFreechart et GanttProject.

Tableau 3.2 Comparaison des algorithmes

Projet	**Algorithme**	**HV**		**DG**		**SP**	
		Médiane	*p-value*	*Médiane*	*p-value*	*Médiane*	*p-value*
JFreechart	NSGA-II	0.6078		0.5312		0.0210	
	NSGA-III	0.3202	>0.05	0.8555	>0.05	0.0208	**<0.05**
	SPEA2	0.3324	>0.05	0.5428	**<0.05**	0.0210	**<0.05**
GanttProject	NSGA-II	0.9157		0.1436		0.1125	
	NSGA-III	0.8967	**<0.05**	0.2406	>0.05	0.1147	**<0.05**
	SPEA2	0.3154	>0.05	0.3220	>0.05	0.2836	>0.05

La p-value est utilisée pour quantifier la significativité statistique entre les algorithmes. Elle indique si la médiane de l'algorithme NSGA-II est statistiquement différente de celle de NSGA-III et de SPEA2 avec un niveau de confiance de 95% ($\alpha = 5\%$). Ce que nous concluons du Tableau 3.2 précédent est que NSGA-II est plus performant que NSGA-III et SPEA2. En effet pour la métrique Hyper volume (HV), NSGA-II est statistiquement différent des deux autres algorithmes pour le projet JFreechart et il est indifférent de NSGA-III pour le deuxième projet. Ce qui signifie

que NSGA-II produit les meilleures (plus grandes) valeurs d'hyper volume. Pour la deuxième métrique distance générationnelle (DG), NSGA-II est statistiquement indifférent de SPEA2 dans le système JFreechart, or que plus la distance générationnelle est petite plus l'algorithme est performant. Donc pour la deuxième métrique aussi NSGA-II est plus performant. Finalement, l'espacement (SP) indique la distance entre les solutions sur le front Pareto, plus cette distance est petite et uniforme plus l'algorithme est performant. Le même cas aussi pour NSGA-II et NSGA-III pour le projet GanttProject. Par contre, pour le projet JFreechart, les 3 algorithmes montrent qu'ils sont statiquement indifférents. En résumé, nous pouvons conclure que NSGA-II surpasse les autres algorithmes pour les projets JFreechart et GanttProject, ce qui confirme notre choix.

3.5.2 Résultats de QR2

Les Tableaux 3.3, 3.4 et 3.5 présentent les résultats de la deuxième question en termes des moyennes de précision, de rappel et de taux d'échec pour chaque projet en utilisant BlobBreak et JDeodorant. Les résultats par détails sont présentés dans l'Annexe I.

- Combinaison de 2 classes :

Pour les combinaisons des deux classes, la performance de BlobBreak dépasse celle de JDeodorant. En effet, pour le projet Ant apache la précision de notre approche est égale à 91.59% et le rappel est égal à 93.12%. Tandis que pour JDeodorant la précision est 69.25% et le rappel est 74.87% avec un taux d'échec de 20%. Pour Xerces, la précision de BlobBreak est 89.20%, le rappel est 87.98% et le taux d'échec est égal à 20%. JDeodorant par contre avait une précision plus faible et égale à 78.63%, un rappel égal à 79.62% et un taux d'échec énorme de 60%. Ce qui signifie que, sur les 5 combinaisons, seulement 2 combinaisons ont été détectées comme classes de trop grande taille. Pour le troisième projet Jfreechart, la précision et le rappel étaient maximales et de l'ordre de 99.42% et 99.23% avec 0 échec. La même chose, JDeodorant n'a aucun échec mais leur précision et rappel sont inférieurs de celle de BlobBreak, 79.11% et 84.04%, respectivement. Pour GanttProject aussi il n'avait pas des cas d'échecs, avec précision

de 81.79% et rappel de 84.11% pour BlobBreak et précision de 68.78% et rappel de 70.83% pour JDeodorant. Finalement pour Rhino, la précision de notre approche était 91.15% et le rappel 70.90% en se comparant à 70.9% de précision et 68.61% de rappel avec une cas d'échec pour JDeodorant.

Tableau 3.3 Précision, rappel et taux d'échec pour la combinaison de 2 classes

Projet	**BlobBreak**			**JDeodorant**		
	Précision%	*Rappel%*	*Taux d'échec%*	*Précision%*	*Rappel%*	*Taux d'échec%*
Ant Apache	91.59	93.12	20	69.25	74.87	20
Xerces	89.20	87.98	20	78.63	79.62	60
Jfreechart	99.42	99.23	0	79.11	84.04	0
GanttProject	81.79	84.11	0	68.78	70.83	0
Rhino	91.15	85.67	0	70.90	68.61	20
Moyenne	**90,635**	**90,02**	**8**	**73,33**	**75,59**	**20**

En moyenne pour tous les projets, la précision de BlobBreak est 90.63% et le rappel est 90% avec un taux d'échec de 8%. Tandis que la précision de JDeodorant est 73.33% et le rappel est 75.59% avec 20% comme taux d'échec.

- **Combinaison de 3 classes :**

Les résultats de combinaisons de 3 classes sont comme suit. Pour le projet Ant apache, la précision de BlobBreak est de 81.75% devant presque la moitié 48.16% pour JDeodorant. Le rappel est de 78.24% pour notre cas par contre le rappel de JDeodorant est plus que la précision et égal à 59.79% avec 20% comme taux d'échec. Pour le deuxième projet Xerces, 75.08% de précision de BlobBreak avec 78.37% de rappel. Pour JDeodorant, nous observons une amélioration par rapport au premier projet avec 65% de précision et presque le même rappel 57.81%, par contre un taux d'échec important de l'ordre de 40%. La précision et le rappel de BlobBreak pour Jfreechart sont presque égaux et de l'ordre de 70%. Hors que pour JDeodorant, toujours leur précision et rappel sont inférieurs à notre approche avec 54.2% et 60.46%, respectivement, en plus d'un taux d'échec de 20%. Pour l'avant-dernier projet GanttProject, la précision et le rappel sont maximales pour notre cas et de l'ordre de 82.85% de précision et 79.59% de rappel et

toujours sans échec. Par contre, JDeodorant a échoué de donner des recommandations avec une précision de 50.54% et un rappel de 60.28%. Enfin, la précision et le rappel étaient 71.08% et 73.89%, respectivement, pour Rhino. En contre partie, JDeodorant a eu 58.85% de précision et 59.81% de rappel avec un cas d'échec.

Tableau 3.4 Précision, rappel et taux d'échec pour la combinaison de 3 classes

Projet	**BlobBreak**			**JDeodorant**		
	Précision%	*Rappel%*	*Taux d'échec%*	*Précision%*	*Rappel%*	*Taux d'échec%*
Ant Apache	81.75	78.24	0	48.16	59.79	20
Xerces	75.08	78.37	0	65	57.81	40
Jfreechart	77	76.79	0	54.2	60.46	20
GanttProject	82.85	79.59	0	50.54	60.28	20
Rhino	71.08	73.89	0	58.85	59.81	20
Moyenne	**77,55**	**77.33**	**0**	**55.35**	**59.63**	**24**

En moyenne pour tous les projets, la précision de BlobBreak est 77.55% et le rappel est 77.33% sans aucun échec. Tandis que la précision de JDeodorant est 55.35% et le rappel est 59.63% avec un taux d'échec de 24%.

- **Combinaison de 4 classes :**

Pour le cas de combinaisons de 4 classes, la précision et le rappel de notre approche ont relativement diminué un peu mais toujours mieux que JDeodorant. Pour Ant apache la précision était 70,81% et le rappel 74.43%. D'une autre coté, JDeodorant a eu une précision d'environ 52.72% et un rappel de 61.83%. La précision pour le projet Xerces était de 65.58% avec un rappel de 69.14% et JDeodorant s'approche un peu avec une précision de 60.29% et un rappel de 57.90%. Pour Jfreechart la précision et le rappel sont presque les mêmes que pour Xerces et de l'ordre de 64.14% et 68.7%, respectivement. Tandis que pour JDeodorant la précision est égale à 52.13% et le rappel égal a 60.85% avec un échec de 40%. Pour GanttProject aussi il n'y avait pas des cas d'échecs avec précision de 66.43% et rappel de 66.14% pour BlobBreak et précision de 38.05% et rappel de 45.31% pour JDeodorant. Finalement pour Rhino, la précision de notre approche était 62% et le rappel 67.24% en se comparant à 33.48% et 38.91% de rappel avec un

cas d'échec pour JDeodorant. En moyenne pour tous les projets, la précision de BlobBreak est

Tableau 3.5 Précision, rappel et taux d'échec pour la combinaison de 4 classes

Projet	**BlobBreak**			**JDeodorant**		
	Précision%	*Rappel%*	*Taux d'échec%*	*Précision%*	*Rappel%*	*Taux d'échec%*
Ant Apache	70.81	74.43	0	52.72	61.83	0
Xerces	65.58	69.14	0	60.29	57.9	0
Jfreechart	64.14	68.70	0	52.13	60.85	40
GanttProject	66.43	66.14	0	38.05	45.31	0
Rhino	62	67.24	0	33.48	38.91	20
Moyenne	**65.79**	**69.13**	**0**	**47.33**	**52.96**	**12**

65.79% et le rappel est 69.13% sans aucun échec. Tandis que la précision de JDeodorant est 47.33% et le rappel est 52.96% avec un taux d'échec de 12%.

La comparaison des performances en fonction de précision et de rappel est importante, mais elle n'approuve pas que la qualité des classes ainsi que les systèmes est améliorée. Dans la question suivante nous allons analyser les résultats de quelques métriques de qualité avant et après la restructuration des classes.

3.5.3 Résultats de QR3

Comme nous avons indiqué dans la section des méthodes d'analyse, pour répondre à cette question nous avons effectué une comparaison entre les métriques de qualité avant et après l'application de refactoring par BlobBreak ainsi que par JDeodorant. Les résultats sont présentés dans le Tableau 3.6 :

Comme le montre le Tableau 3.6, le couplage calculé à l'aide de métrique CBO augmente pour tous les projets après l'application de refactoring que ce soit par BlobBreak ou par JDeodorant. Ceci a fait du sens parce qu'en appliquant l'opération d'extraction des classes, de nouvelles classes seront crées et le nombre des liaisons entre les classes va augmenter. Nous avons calculé la moyenne de CBO pour tous les projets. Nous avons trouvé que la moyenne de CBO avant le refactoring était égale à 20.78. Après le refactoring, la moyenne de CBO a passé à 24.72 pour le

Tableau 3.6 Comparaison entre la qualité avant et après l'application de refactoring par JDeodorant et BlobBreak pour tous les projets

Projets	Avant Refactoring			JDeodorant			BlobBreak		
	CBO	*LCOM*	*CAMC*	*CBO*	*LCOM*	*CAMC*	*CBO*	*LCOM*	*CAMC*
Ant Apache	12	880.6	0.1	17.6	313.75	0.18	16.1	207.41	0.28
Xerces	21.9	1484.5	0.16	26.6	569.06	0.34	28.4	340.57	0.37
Jfreechart	8.8	786.6	0.13	13	668.58	0.19	11.5	284.6	0.25
GanttProject	29.6	2670.9	0.09	32.7	1229.28	0.25	31.9	642.4	0.38
Rhino	31.6	2954.4	0.13	33.7	1505.56	0.25	39.6	854.59	0.32
Moyenne (Tous les projets)	**20.78**	**1755.4**	**0.126**	**24.72**	**857.24**	**0.24**	**25.5**	**465,91**	**0.32**

cas de JDeodorant et à 25.5 pour le cas de BlobBreak. La valeur de CBO pour notre approche est supérieure à celle de JDeodorant parce que BlobBreak a extrait plus de classes que JDeodorant à fin de mieux corriger les classes de trop grande taille.

En terme de cohésion, nous avons utilisé deux métriques pour évaluer nos résultats qui sont LCOM et CAMC. Dans le cas de LCOM, plus la valeur est élevée plus la cohésion est faible. Par contre, pour CAMC plus la valeur est faible plus la cohésion est faible. Donc, d'après le Tableau 3.6, nous constatons que JDeodorant améliore la cohésion mais BlobBreak l'améliore plus pour toutes les projets. Nous avons calculé les moyennes de LCOM et CAMC pour tous les projets. Les résultats montrent que la cohésion avant l'application de refactoring était très faible. En effet la valeur moyenne de LCOM avant refactoring est 1755.4 et la valeur moyenne de CAMC est 0.12. Après l'application de refactoring en utilisant JDeodorant la valeur moyenne de LCOM a passé à 857,24 et CAMC à 0,24. Tandis qu'avec BlobBreak, la cohésion s'est plus améliorée et égale presque la moitié de LCOM pour le cas de JDeodorant (467,25) et la valeur moyenne de CAMC aussi est supérieure à celle de JDeodorant et égale à 0,32. Les résultats par détails de 5 projets sont présentés dans l'Annexe II. Dans la section discussion, nous allons discuter plus en détail un seul projet pour voir l'évolution de ces métriques avant et après le refactoring.

3.5.4 Résultats de QR4

Les résultats de la quatrième question sont présentés dans le Tableau 3.7. Pour la première paire

Tableau 3.7 Impact de chaque heuristique sur la précision et le rappel

Classes		Struct+Sem		Struct+Co-changement		Sem+Co-changement		Struct+Sem+Co-changement	
		Précison%	*Rappel%*	*Précison%*	*Rappel%*	*Précison%*	*Rappel%*	*Précison%*	*Rappel%*
1	CategoryPointerAnnotation.java AxisCollection.java	84,93	90,39	82,03	85,71	83,33	84,44	100	100
2	TextAnnotation.java FlowArrangement.java	84	89,76	79,82	67,85	98,33	96,66	100	100
3	RectangleInsets.java DefaultShadowGenerator.java	92,68	78,57	62,34	60,51	65,12	61,66	97,14	96,19
4	AxisEntity.java SunJPEGEncoderAdapter.java	71,87	68,18	88,23	71,42	71,87	68,18	100	100
5	ColorBlock.java AxisCollection.java	100	100	100	100	85	81,25	100	100
Moyenne		**86,69**	**85,38**	**82,48**	**77,09**	**80,73**	**78,43**	**99,42**	**99,23**

de classes, la précision et le rappel de la combinaison de dépendances structurelles et sémantiques sont 84.93% et 90.39%, respectivement. Pour les deux autres combinaisons, structurelle plus le co-changement et les dépendances sémantiques plus le co-changement, les pourcentages sont un peu proches de la première combinaison et de l'ordre de 82.03% et 83.33% pour la précision et 85.71% et 84.44% de rappel, respectivement. Alors que la combinaison des trois heuristiques mène à une précision et un rappel de 100%.

On passe ensuite aux deuxièmes paires des classes, pour l'ensemble de deux heuristiques structurelle et sémantique, la précision était de l'ordre de 84% et le rappel est 89.76%. La précision et le rappel de la combinaison structurelle plus le co-changement sont, 79.82% et 96.66%, respectivement. Puis pour la combinaison sémantique plus le co-changement, la précision est 98.33% et le rappel est 96.66%. La précision et le rappel maximales sont pour le cas de l'utilisation de trois heuristiques ensemble (100%).

Pour la troisième paire de classes, la précision et le rappel sont 92.68% et 78.57%, respectivement, pour le cas d'utilisation des heuristiques structurelle plus sémantique. Les résultats se dégradent quand nous utilisons les heuristiques, structurelle plus le co-changement et la sémantique de membres plus le co-changement, avec 62.34% de précision et 60.51% de rappel pour le premier cas et 65.12% de précision et 61.66% de rappel pour le deuxième cas. Nous observons une amélioration importante des résultats, en utilisant, la combinaison de trois heuristiques ensemble avec 97.14% de précision et 96.19% de rappel.

Pour l'avant-dernier cas, la précision et le rappel sont élevés et de l'ordre de 100% quand nous avons utilisé les trois heuristiques ensemble. Alors que pour la combinaison d'heuristiques, structurelle plus sémantique, et sémantique plus co-changement la précision et le rappel sont les mêmes, 71.87% et 68.18%, respectivement. La précision et le rappel de combinaison structurelle plus le co-changement sont plus élevés et de l'ordre de 88.23% et 71.42%.

Finalement, la dernière paire des classes a eu une précision et un rappel maximales, 100%, pour ces trois cas de figures : la combinaison des heuristiques structurelle et sémantique, la combinaison des heuristiques structurelle et le co-changement et la combinaison des tous les heuristiques utilisés dans notre approche. Alors que pour le cas de l'utilisation que des heuristiques, sémantique plus le co-changement, l'outil a échoué d'extraire les classes telles qu'elles étaient avant avec une précision de 85% et un rappel de 81.25%. Dans la section discussion, nous allons détailler l'impact de chaque heuristique avec des exemples.

3.6 Discussion

Nos résultats expérimentaux démontrent que BlobBreak surpasse de manière significative JDeodorant.

Pour les résultats de la deuxième question, nous observons que notre outil a une haute scalabilité en terme de taille des projets. En effet, les résultats ont montré que, quelle que soit la taille de projet, BlobBreak a eu des pourcentages de précision et de rappel élevés. Aussi, la précision et le rappel de notre approche dépassent celles de JDeodorant pour toutes les combinaisons pour plusieurs raisons. Premièrement, JDeodorant ne considère que les liaisons structurelles entre les membres de la classe. Malgré qu'il existe des cas là ou deux membres non aucune relation structurelle entre eux, mais ils sont similaires sémantiquement. Dans plusieurs cas, des méthodes se modifient dans plusieurs reprises par le développeur en même temps, ces membres doivent être extraites ensemble mais JDeodorant les sépare. Aussi, JDeodorant extrait des entités de la classe dans une nouvelle classe, mais il fait une duplication des méthodes dans la classe d'origine et la classe extraites. C'est-à-dire, il laisse les méthodes dans la classe d'origine ainsi que dans

la classe extraite. Les méthodes de la classe extraite sont celles responsables de traitement, par contre le rôle des méthodes dans la classe d'origine est de les appeler. En d'autres termes, JDeodorant applique le refactoring "Move Method". Ce qui augmente le nombre des méthodes dans la classe d'origine par rapport à notre approche et qui contribue par conséquent à diminuer sa précision.

Passant par la suite aux résultats de la troisième question, qui ont démontré que BlobBreak améliore la qualité des classes ainsi que des projets en terme de cohésion plus que JDeodorant. Si nous prenons le projet Ant Apache, nous trouvons que dans 90% des cas, les classes sont plus cohésives avec BlobBreak. Le seul cas là où les valeurs moyennes de cohésion, en utilisant JDeodorant, surpassent note approche est pour la classe "`AntClassLoader`". JDeodorant a extrait 2 classes avec des valeurs moyennes de CAMC 0.1 pour la classe originale et 0.41, 0.31 pour les deux classes extraites, respectivement. Pour BlobBreak la cohésion de la classe originale est 0.21 et la valeur de CAMC de deux classes extraites sont 0.19 et 0.25, respectivement. Ce qui est claire est que notre outil a amélioré la cohésion pour la classe originale sauf que JDeodorant a extrait deux classes plus cohésives que les deux classes extraites par notre approche. Ce qui fait que la moyenne pour JDeodorant soit supérieure que BlobBreak. Donc la partition des classes dans ce cas est plus cohésive. Ici il faut signaler que lorsque nous avons utilisé PMD une autre fois après la restructuration de la classe avec BlobBreak, la classe originale et les deux classes extraites n'apparaissent pas comme des classes de trop grande taille. Dans tous les autres cas, les mesures de métrique ont montré que l'utilisation de BlobBreak aboutit à améliorer la cohésion des classes et de tout le projet. Ce que nous pouvons conclure de la dernière question est qu'il faut prendre en compte de trois heuristiques structurelles, sémantiques et le co-changement dans le problème de restructuration de classe pour avoir une précision et un rappel maximales. Avec juste deux heuristiques, que ce soit la combinaison, la précision varie de 62.34% à 100% et le rappel varie de 60.51% et 100%. Il existe des cas là où nous observons la nécessité d'une heuristique pour avoir des résultats plus précis. Prenant l'exemple de la combinaison des dépendances structurelles plus le co-changement avec le pair des classes "`TextAnnotation.java`" et "`FlowArrangement.java`" là où nous trouvons

Tableau 3.8 Comparaison entre la qualité avant et après l'application de refactoring par JDeodorant et notre approche pour Ant Apache

Classes	Avant la restructuration de la classe			Restructuration de la classe avec JDeodorant			Restructuration de la classe Avec notre approche		
	CBO	*LCOM*	*CAMC*	*CBO*	*LCOM*	*CAMC*	*CBO*	*LCOM*	*CAMC*
AntClassLoader	11	1584	0.001	14	440.33	0.27	14	442.33	0.220633
ComponentHelper	11	78	0.13	16	260.25	0.28	14	170.33	0.29
FormatterElement	8	117	0.10	13	30.5	0.11	9	11.5	0.18
AbstractCvsTask	10	981	0.11	16	48	0.14	16	151.33	0.24
ntrospectionHelper	28	2626	0.14	50	442	0.16	42	438	0.17
Pvcs	10	528	0.12	10	528	0.12	11	328	0.50
SubAnt	15	420	0.06	23	204.5	0.10	21	69	0.18
WeblogicDeploymentTool	11	413	0.12	11	413	0.12	12	158.5	0.27
RecorderEntry	5	261	0.12	6	108	0.22	7	85.5	0.49
PropertyHelper	11	1088	0.13	17	227	0.25	15	219.66	0.30
Moyenne	**12**	**880.6**	**0.1**	**17.6**	**313.75**	**0.18**	**16.1**	**207.41**	**0.28**

les méthodes "`arrangeFR(BlockContainer,Graphic2D,RectangleConst)`" et "`arrangeFN(BlockContainer,Graphic2D,RectangleConst)`". Ces méthodes sont sémantiquement très similaires mais l'outil ne les extrait pas ensemble. Dans ce cas nous n'avons pas pris en compte les dépendances sémantiques, ce qui explique ces résultats. Si nous prenons la combinaison des dépendances structurelles et sémantiques et le même exemple de classes. Nous trouvons que les deux méthodes "`setFont()`" et la méthode "`setText()`", ont été modifiées par le développeur plusieurs fois ensemble, mais lors de la restructuration de la classe elles ont été classées chacune dans une classe séparé. Il faut préciser aussi que ces deux méthodes n'ont aucune dépendance structurelle entre eux, ni sémantique. Ce qui explique qu'il faut prendre en considération aussi le co-changement entre les méthodes pour restructurer les classes correctement.

3.7 Menaces à la validité

Nous explorons dans cette section les facteurs pouvant biaiser notre étude expérimentale. Ces facteurs peuvent être classés en trois catégories : validité de construction, validité interne et validité externe.

3.7.1 La validité de construction

La validité de construction concerne la relation entre la théorie et ce qui est observé. Une menace possible est liée à l'absence de travaux similaires utilisant des algorithmes basés sur la recherche métaheuristique pour l'extraction des classes de trop grande taille, donc nous avons évalué notre approche et la comparer à une solution déterministe, JDeodorant.

Une autre menace réside dans le fait que nous donnons le même poids aux trois heuristiques quand nous calculons nos fonctions objectifs. Ça peut dégrader nos résultats dans des cas spécifiques là où les classes sont mal conçues par le développeur. C'est-à-dire, il n'existe pas des similarités sémantiques entre les membres de la classe, ainsi que le développeur effectue des changements de code source arbitrairement. Pour cela, nous planifions dans nos travaux futurs d'optimiser nos fonctions objectifs par attribuer un coefficient à chaque heuristique. Ces coefficients seront calculés après une analyse profonde de la classe pour savoir quelle heuristique est à prioriser.

3.7.2 La validité interne

La validité interne concerne la relation de cause à effet entre le traitement et le résultat. Une menace possible pour la validité interne réside dans l'utilisation d'algorithmes stochastiques. Pour contourner cette menace, notre étude expérimentale est réalisée sur la base de nombreuses simulations indépendantes pour chaque instance problématique.

Une autre limite est liée à la performance de notre approche. Le temps d'exécution dépend principalement de calcul des heuristiques, surtout pour le calcul de co-changement. Ce temps est long parce que l'outil analyse tous les« commits » pour trouver celles de la classe de trop grande taille, ensuite il extrait les modifications de la classe, et il calcule les méthodes qui se modifient ensemble en comparant les codes sources.

3.7.3 La validité externe

La validité externe fait référence à la généralisation de nos résultats. Dans cette étude, nous avons effectué nos expériences sur cinq systèmes du domaine du logiciel libre, largement utilisés, de tailles différentes et de domaines d'applications différentes, comme décrit dans le tableau 3.1. Cependant, nous ne pouvons pas affirmer que nos résultats peuvent être généralisés avec d'autres applications et d'autres langages de programmation. Des reproductions futures de cette étude sont nécessaires pour confirmer la généralisation de nos résultats.

Une autre menace est que, malgré que les résultats sont encourageants, il faut impliquer les développeurs dans l'évaluation pour s'assurer que notre outil est utile et que les résultats sont applicables. Nous planifions dans nos travaux futurs d'évaluer notre approche avec des développeurs indépendants du milieu industriel.

3.8 Conclusion

Dans ce chapitre, nous avons cité nos questions de recherches ainsi que les systèmes que nous avons étudié et les méthodes d'analyse que nous avons utilisé. Ensuite, nous avons présenté et discuté les résultats obtenus et les menaces qui affectent la validité de nos expérimentations.

CONCLUSION ET RECOMMANDATIONS

Le travail présenté dans ce mémoire porte sur l'automatisation du processus de restructuration de classes de trop grande taille, afin d'aider les ingénieurs chargés de la maintenance et d'améliorer la qualité de logiciel. À cette fin, nous avons appliqué une technique basée sur la recherche métaheuristique (SBSE) qui s'est révélée être un moyen pratique et efficace pour résoudre plusieurs problèmes d'ingénierie logicielle. Pour ce faire, nous avons utilisé un algorithme d'optimisation multi-objectif NSGA-II pour résoudre le problème et générer les recommandations optimales d'extraction des classes de trop grande taille. Notre approche combine différents aspects sur les dépendances statiques, sémantiques et le co-changement. Une solution optimale doit trouver le meilleur compromis entre les objectifs suivants (i) maximiser la cohésion des classes extraites, (ii) minimiser les dépendances externes entre ces classes. Par conséquent, améliorer également la conception générale du système et sa complexité.

Nous avons évalué l'efficacité de notre approche, intitulée BlobBreak, en utilisant cinq systèmes open source. Nous avons commencé par évaluer les performances de BlobBreak et JDeodorant, un outil existant pour l'extraction des classes de trop grande taille. Nous avons choisi, aléatoirement, des classes qui ne sont pas des Blobs. Puis nous avons formé cinq combinaisons de 2 classes, cinq combinaisons de 3 classes et cinq combinaisons de 4 classes pour chaque projet. Ensuite, nous avons utilisé BlobBreak et JDeodorant pour savoir si les deux outils sont capables de restructurer les classes et de les rendre à leurs formes originales.Ainsi, les résultats ont montré que BlobBreak a atteint les meilleures performances en terme de précision et de rappel avec en moyenne, 90.63% de précision, 90% de rappel et 8% d'échec pour la première combinaison. 77.55% de précision et 77.33% de rappel en moyenne avec 0 échec pour la deuxième combinaison de 3 classes. Pour la combinaison de 4 classes la précision est 65.79%, le rappel 69.13% en moyenne avec 0 échec. Les résultats obtenus confirment l'efficacité de BlobBreak par rapport à l'approche existante JDeodorant. Ainsi, nous avons effectué une autre expérimentation pour évaluer la qualité des classes traitées, en terme de cohésion et de couplage, avant et après le refactoring.

Nous avons aussi comparé ces mesures de qualité avec JDeodorant. Les résultats ont montré que BlobBreak améliore la qualité des classes plus que JDeodorant et ceci pour tous les projets étudiés. Finalement, nous avons aussi étudié l'impact de chaque heuristique sur nos résultats. Nous avons trouvé que l'utilisation de trois heuristiques ensemble donne les meilleurs résultats en termes de précision et de rappel.

Travaux futurs :

Certaines directions de travaux futures peuvent être explorées. Premièrement, notre approche utilise l'historique des changements de code pour recommander des propositions de restructuration de classe. Ces changements sont collectés à partir de système de contrôle de version Git. Comme travaux futurs, nous allons travailler sur l'extension de notre approche afin de prendre en charge l'historique des modifications collectées à partir d'autres Systèmes de Contrôle de Version. Deuxièmement, nous planifions de recommander automatiquement des noms de classes extraites qui sont dérivés du vocabulaire utilisé. Nous planifions aussi d'optimiser nos fonctions objectifs par l'attribution d'un coefficient à chaque heuristique. Ces coefficients vont prioriser une heuristique par rapport à un autre selon les dépendances existantes dans la classe. Une autre extension intéressante est d'ajouter l'option d'appliquer les recommandations de refactoring proposées par le « plugin », automatiquement. Nous visons également d'évoluer les fonctionnalités de notre « plugin » pour qu'il s'adapte avec d'autres IDE ainsi qu'avec d'autres langages de programmation. Finalement, nous allons effectuer une évaluation avec des développeurs indépendants pour s'assurer de l'utilité de notre approche.

ANNEXE I

LES RÉSULTATS DE LA DEUXIÈME QUESTION EN DÉTAIL

Dans les tableaux qui suivent, nous allons présenter en détails les résultats de la deuxième question en termes de précision, de rappel et de taux d'échec dans les cas de combinaisons de 2 classes, de 3 classes et de 4 classes pour les 5 projets.

1. Combinaison de 2 classes

Tableau-A I-1 Combinaison de 2 classes

Projets		Classe	BlobBreak		JDeodorant	
			Precision%	Rappel%	Precision%	Rappel%
Rhino	1	org.mozilla.javascript.commonjs.module.Require.java org.mozilla.javascript.EcmaError.java	86,74	82,36	83,58	80,25
	2	org.mozilla.javascript.typedarrays.NativeDataView.java org.mozilla.javascript.optimizer.Optimizer.java	76,35	67,43	37,5	50
	3	org.mozilla.javascript.Arguments.java org.mozilla.javascript.commonjs.module.ModuleScript.java	97,22	85,71	92,55	82,67
	4	org.mozilla.javascript.BoundFunction.java org.Mozilla.javascript.ContinuationPending.java	95,45	92,85	0	0
	5	org.mozilla.javascript.regexp.SubString.java org.mozilla.javascript.optimizer.ClassCompiler.java	100	100	70	61,53
		Moyenne%	91,152	85,67	70,9075	68,6125
		Taux d'échec%	0		20	
Xerces	1	org.apache.xerces.impl.dv.XSFacets.java org.apache.xerces.dom.events.EventImpl.java	89,06	83,33	0	0
	2	org.apache.xerces.impl.xs.AttributePSVImpl.java org.apache.xerces.impl.io.ASCIIReader.java	100	100	87,14	85,71
	3	org.apache.xerces.impl.xs.opti.DefaultDocument.java org.apache.xerces.impl.xs.models.CMNodeFactory.java	72,3	74,85	70,12	73,54
	4	org.apache.xml.serialize.IndentPrinter.java org.apache.xml.serialize.SerializerFactory.java	0	0	0	0
	5	org.apache.xerces.util.DOMInputSource.java org.apache.xerces.xni.QName.java	95,45	93,75	0	0
		Moyenne%	89,2025	87,9825	78,63	79,625
		Taux d'échec%	20		60	
Ant Apache	1	org.apache.tools.ant.taskdefs.compilers.DefaultCompilerAdapter.java org.apache.tools.ant.taskdefs.condition.Contains.java	96,25	98,34	95,83	98,12
	2	org.apache.tools.ant.taskdefs.condition.ConditionBase.java org.apache.tools.ant.taskdefs.condition.AntVersion.java	86,11	90,38	30,76	46,15
	3	org.apache.tools.ant.taskdefs.cvslib.ChangeLogTask.java org.apache.tools.ant.taskdefs.email.Message.java	84,01	83,79	58,33	55,21
	4	org.apache.tools.zip.FallbackZipEncoding.java org.apache.tools.zip.JarMaker.java	0	0	0	0
	5	org.apache.tools.ant.listener.BigProjectLogger.java org.apache.tools.ant.input.DefaultInputHandler.java	100	100	92,105	100
		Moyenne%	91,5925	93,1275	69,25625	74,87
		Taux d'échec%	20		20	

Tableau-A I-2 Combinaison de 2 classes (Suite)

Projets		Classe	BlobBreak		JDeodorant	
			Precision%	Rappel%	Precision%	Rappel%
Jfreechart	1	org.jfree.chart.annotations.CategoryPointerAnnotation.java org.jfree.chart.axis.AxisCollection.java	100	100	61,61	66,66
	2	org.jfree.chart.annotations.TextAnnotation.java org.jfree.chart.block.FlowArrangement	100	100	69,04	67,5
	3	org.jfree.chart.ui.RectangleInsets.java org.jfree.chart.util.DefaultShadowGenerator.java	97,1456	96,19	96,32	98,54
	4	org.jfree.chart.entity.AxisEntity.java org.jfree.chart.encoders.SunJPEGEncoderAdapter.java	100	100	84,21	87,5
	5	org.jfree.chart.block.ColorBlock.java org.jfree.chart.AxisCollection	100	100	84,37	100
		Moyenne%	99,42912	99,238	79,11	84,04
		Taux d'échec%	0		0	
GanttProject	1	net.sourceforge.ganttproject.chart.ChartModelBase.java net.sourceforge.ganttproject.calendar.AlwaysWorkingTimeCalendarImpl.java	67,74	69,54	48,98	45,23
	2	net.sourceforge.ganttproject.chart.ChartModelImpl.java net.sourceforge.ganttproject.document.AbstractDocument.java	87,24	89,75	46,85	43,95
	3	net.sourceforge.ganttproject.export.ConsoleUIFacade.java net.sourceforge.ganttproject.filter.ExtensionBasedFileFilter.java	64,015	71,26	91,25	75
	4	net.sourceforge.ganttproject.calendar.CalendarActivityImpl.java net.sourceforge.ganttproject.FacadeInvalidator.java	90	90	73,52	90
	5	net.sourceforge.ganttproject.TreeTableCellEditorImpl.java net.sourceforge.ganttproject.calendar.GPCalendarBase.java	100	100	83,33	100
		Moyenne%	81,799	84,11	68,786	70,836
		Taux d'échec%	0		0	

2. Combinaison de 3 classes

Tableau-A I-3 Combinaison de 3 classes

Projets		Classe	BlobBreak		JDeodorant	
			Precision%	Rappel%	Precision%	Rappel%
Rhino	1	1/org.mozilla.javascript.BoundFunction,java	93,58	85,27	0	0
		2/org.Mozilla.javascript.ContinuationPending,java				
		3/org.Mozilla.javascript.commonjs.module.provider.ModuleSource,java				
	2	1/org.mozilla.javascript.ConsString.java	50	50	57,14	48,8
		2/org.mozilla.javascript.EcmaError.java				
		3/org.mozilla.javascript.ModuleScript.java				
	3	1/org.mozilla.javascript.regexp.SubString.java	46,03	60	53,29	57,14
		2/org.mozilla.javascript.optimizer.ClassCompiler.java				
		3/org.mozilla.javascript.serialize.ScriptableInputStream.java				
	4	1/org.mozilla.javascript.typedarrays.NativeDataView.java	91,2	93,87	81,31	86,11
		2/org.mozilla.javascript.optimizer.Optimizer.java				
		3/org.mozilla.javascript.json.JsonParser				
	5	1/org.mozilla.javascript.commonjs.module.Require.java	74,6	80,34	43,67	47,22
		2/org.mozilla.javascript.ClassCache.java				
		3/org.mozilla.javascript.EcmaError.java				
		Moyenne%	71,082	73,896	58,8525	59,8175
		Taux d'échec%	0		20	
Xerces	1	1/org.apache.xerces.impl.dtd.DTDGrammarBucket.java	69,31	76,6	59,16	55,83
		2/org.apache.xerces.dom.events.EventImpl.java				
		3/org.apache.xerces.impl.dv.DatatypeException.java				
	2	1/org.apache.xml.serialize.IndentPrinter.java	46	57,14	54,9	47,61
		2/org.apache.xml.serialize.SerializerFactory.java				
		3/org.apache.xerces.xpointer.XPointerErrorHandler.java				
	3	1/org.apache.xerces.util.DOMInputSource.java	100	100	0	0
		2/org.apache.xerces.xni.QName.java				
		3/org.apache.xerces.impl.xs.util.ShortListImpl.java				
	4	1/org.apache.xerces.impl.dv.XSFacets.java	96,5	98	80,95	70
		2/org.apache.xerces.dom.events.EventImpl.java				
		3/org.apache.xerces.impl.io.ASCIIReader.java				
	5	1/org.apache.xerces.impl.xs.AttributePSVImpl.java	63,6	60,12	0	0
		2/org.apache.xerces.impl.io.ASCIIReader.java				
		3/org.apache.xerces.impl.xs.models.CMNodeFactory.java				
		Moyenne%	75,082	78,372	65,00333333	57,81333333
		Taux d'échec%	0		40	
Ant	1	1/org.apache.tools.ant.BuildException.java	88,09	78,57	45,55	66
		2/org.apache.tools.ant.input.InputRequest.java				
		3/org.apache.tools.ant.listener.Log4jListener.java				
	2	org.apache.tools.zip.FallbackZipEncoding.java	86,53	72,72	41,97	48,48
		org.apache.tools.zip.JarMaker.java				
		org.apache.tools.ant.util.regexp.JakartaOroMatcher.java				
	3	org.apache.tools.ant.listener.BigProjectLogger.java	98,14	97,54	55,12	66,66
		org.apache.tools.ant.input.DefaultInputHandler.java				
		org.apache.tools.ant.taskdefs.cvslib.RCSFile.java				
	4	1/org.apache.tools.ant.taskdefs.condition.Contains.java	71,21	75	0	0
		2/org.apache.tools.ant.taskdefs.cvslib.RCSFile.java				
		3/org.apache.tools.ant.taskdefs.email.EmailAddress.java				
	5	1/org.apache.tools.ant.taskdefs.condition.ConditionBase.java	64,82	67,37	50	58,02
		2/org.apache.tools.ant.taskdefs.optional.RenameExtension.java				
		3/org.apache.tools.ant.taskdefs.cvslib.ChangeLogTask.java				
		Moyenne%	81,758	78,24	48,16	59,79
		Taux d'échec%	0		20	

Tableau-A I-4 Combinaison de 3 classes(Suite)

Projets		Classe	BlobBreak		JDeodorant	
			Precision%	Rappel%	Precision%	Rappel%
Jfreechart	1	1/org.jfree.chart.entity.AxisEntity,java	51,28	56,94	45,16	57,14
		2/org.jfree.chart.encoders.SunJPEGEncoderAdapter,java				
		3/org.jfree.chart.editor.ChartEditorManager,java				
	2	1/org,jfree,chart,block,ColorBlock,java	95,83	84,34	0	0
		2/org.jfree.chart.StrokeMap.java				
		3/org.jfree.chart.axis.DateTick.java				
	3	1/org.jfree.chart.title.DateTitle.java	75	63,33	39,5	51,51
		2/org.jfree.chart.text.TextBlock.java				
		3/org.jfree.data.statistics.HistogramBin.java				
	4	1/org.jfree.chart.annotations.CategoryPointerAnnotation.java	82,53	94,59	71,54	92,24
		2/org.jfree.chart.axis.AxisCollection.java				
		3/org.jfree.chart.ChartMouseEvent.java				
	5	1/org.jfree.chart.annotations.TextAnnotation.java	80,36	84,75	60,6	40,96
		2/org.jfree.chart.block.FlowArrangement.java				
		3/org.jfree.chart.ui.RectangleInsets.java				
		Moyenne%	77	76,79	54,2	60,4625
		Taux d'échec%	0		20	
GanttProject	1	1/net.sourceforge.ganttproject.calendar.CalendarActivityImpl.java	96,74	85,56	44,44	60
		2/net.sourceforge.ganttproject.FacadeInvalidator.java				
		3/net.sourceforge.ganttproject.gui.GanttMetalTheme.java				
	2	1/net.sourceforge.ganttproject.TreeTableCellEditorImpl.java	72,87	74,9	68,72	68,72
		2/net.sourceforge.ganttproject.calendar.GPCalendarBase.java				
		3/net.sourceforge.ganttproject.filter.ExtensionBasedFileFilter.java				
	3	1/net.sourceforge.ganttproject.GPLoggertest.java	93,93	91,06	43,2	56
		2/net.sourceforge.ganttproject.calendar.GanttDaysOff.java				
		3/net.sourceforge.ganttproject.gui.options.ExportSettingsPanel.java				
	4	1/net.sourceforge.ganttproject.chart.BottomUnitLineRendererImpl.java	87,5	75	45,83	56,41
		2/net.sourceforge.ganttproject.calendar.AlwaysWorkingTimeCalendarImpl.java				
		3/net.sourceforge.ganttproject.document.FileDocument.java				
	5	1/net.sourceforge.ganttproject.chart.ChartModelImpl.java	63,21	71,46	0	0
		2/net.sourceforge.ganttproject.document.AbstractDocument.java				
		3/net.sourceforge.ganttproject.export.ConsoleUIFacade.java				
		Moyenne%	82,85	79,596	50,5475	60,2825
		Taux d'échec%	0		20	

3. Combinaison de 4 classes

Tableau-A I-5 Combinaison de 4 classes

Projets		Classes	BlobBreak		JDeodorant	
			Precision%	Rappel%	Precision%	Rappel%
Rhino	1	1/org.mozilla.javascript.BoundFunction.java	83,33	72,61	0	0
		2/org.Mozilla.javascript.ContinuationPending.java				
		3/org.Mozilla.javascript.commonjs.module.provider.ModuleSource.java				
		4/org.mozilla.javascript.commonjs.module.provider.SoftCachingModuleScriptProvider.java				
	2	1/org.mozilla.javascript.ConsString.java	88,09	93,75	58,12	48,02
		2/org.mozilla.javascript.EcmaError.java				
		3/org.mozilla.javascript.ModuleScript.java				
		4/org.mozilla.javascript.commonjs.module.provider.UrlModuleSourceProvider.java				
	3	1/org.mozilla.javascript.regexp.SubString.java	34,099	50	31,25	50
		2/org.mozilla.javascript.optimizer.ClassCompiler.java				
		3/org.mozilla.javascript.serialize.ScriptableInputStream.java				
		4/org.mozilla.javascript.typedarrays.Conversions.java				
	4	1/org.mozilla.javascript.typedarrays.NativeDataView.java	43,1	52,85	12,31	20,12
		2/org.mozilla.javascript.optimizer.Optimizer.java				
		3/org.mozilla.javascript.json.JsonParser				
		4/org.mozilla.javascript.commonjs.module.provider.ModuleSourceProviderBase.java				
	5	1/org.mozilla.javascript.commonjs.module.Require.java	61,4	67,02	32,26	37,5
		2/org.mozilla.javascript.ClassCache.java				
		3/org.mozilla.javascript.EcmaError.java				
		4/org.mozilla.javascript.Arguments.java				
		Moyenne%	62,0038	67,246	33,485	38,91
		Taux D'échec%	0		20	
Xerces	1	1/org.apache.xerces.impl.dtd.DTDGrammarBucket.java	66,5	63,75	64,51	64
		2/org.apache.xerces.dom.events.EventImpl.java				
		3/org.apache.xerces.impl.dv.DatatypeException.java				
		4/org.apache.xerces.impl.dv.util.ByteListImpl.java				
	2	1/org.apache.xml.serialize.IndentPrinter.java	82	85,03	62,76	65
		2/org.apache.xml.serialize.SerializerFactory.java				
		3/org.apache.xerces.xpointer.XPointerErrorHandler.java				
		4/org.apache.xerces.dom.events.MutationEventImpl.java				
	3	1/org.apache.xerces.util.DOMInputSource.java	52,36	57,41	52,36	36,36
		2/org.apache.xerces.xni.QName.java				
		3/org.apache.xerces.impl.xs.util.ShortListImpl.java				
		4/org.apache.xerces.impl.dtd.models.CMNode.java				
	4	1/org.apache.xerces.impl.dv.XSFacets.java	71,32	76,58	63,29	57,91
		2/org.apache.xerces.dom.events.EventImpl.java				
		3/org.apache.xerces.impl.dv.dtd.DTDDVFactoryImpl.java				
		4/org.apache.xerces.impl.io.ASCIIReader.java				
	5	1/org.apache.xerces.impl.xs.AttributePSVImpl.java	55,73	62,95	58,55	66,25
		2/org.apache.xerces.impl.io.ASCIIReader.java				
		3/org.apache.xerces.impl.xs.opti.DefaultDocument.java				
		4/org.apache.xerces.impl.xs.models.CMNodeFactory.java				
		Moyenne%	65,582	69,144	60,294	57,904
		Taux D'échec%	0		0	
Ant	1	1/org.apache.tools.ant.BuildException.java	84,51	80,9	57,14	75
		2/org.apache.tools.ant.input.InputRequest.java				
		3/org.apache.tools.ant.listener.Log4jListener.java				
		4/org.apache.tools.ant.launch.launcher.java				
	2	1/org.apache.tools.zip.FallbackZipEncoding.java	62,75	59,09	45,19	49,82
		2/org.apache.tools.zip.JarMaker.java				
		3/org.apache.tools.ant.util.regexp.JakartaOroMatcher.java				
		4/org.apache.tools.ant.util.facade.FacadeTaskHelper.java				
	3	1/org.apache.tools.ant.listener.BigProjectLogger.java	85,16	89,75	63,79	75
		2/org.apache.tools.ant.input.DefaultInputHandler.java				
		3/org.apache.tools.ant.taskdefs.cvslib.RCSFile.java				
		4/org.apache.tools.ant.ProjectHelperRepository.java				
	4	1/org.apache.tools.ant.taskdefs.compilers.DefaultCompilerAdapter.java	49,16	67,5	37,5	50
		2/org.apache.tools.ant.taskdefs.condition.Contains.java				
		3/org.apache.tools.ant.taskdefs.cvslib.RCSFile.java				
		4/org.apache.tools.ant.taskdefs.email.EmailAddress.java				
	5	1/org.apache.tools.ant.taskdefs.condition.ConditionBase.java	72,5	74,91	60	59,37
		2/org.apache.tools.ant.taskdefs.optional.RenameExtension.java				
		3/org.apache.tools.ant.taskdefs.cvslib.ChangeLogTask.java				
		4/org.apache.tools.ant.taskdefs.email.Message.java				
		Moyenne%	70,816	74,43	52,724	61,838
		Taux D'échec%	0		0	

Tableau-A I-6 Combinaison de 4 classes(Suite)

Projets		Classes	BlobBreak		JDeodorant	
			Precision%	Rappel%	Precision%	Rappel%
Jfreechart	1	1/org.jfree.chart.entity.AxisEntity.java	59,21	51,62	36	43
		2/org.jfree.chart.encoders.SunJPEGEncoderAdapter.java				
		3/org.jfree.chart.editor.ChartEditorManager.java				
		4/org.jfree.chart.imagemap.ImageMapUtils.java				
	2	1/org,jfree,chart,block,ColorBlock,java	65	68,75	0	0
		2/org.jfree.chart.StrokeMap.java				
		3/org.jfree.chart.axis.DateTick.java				
		4/org.jfree.chart.editor.ChartEditorManager.java				
	3	1/org.jfree.chart.title.DateTitle.java	64,8	75	0	0
		2/org.jfree.chart.text.TextBlock.java				
		3/org.jfree.data.statistics.HistogramBin.java				
		4/org.jfree.chart.plot.MeterInterval.java				
	4	1/org.jfree.chart.annotations.CategoryPointerAnnotation.java	62,31	76,14	54,16	75
		2/org.jfree.chart.axis.AxisCollection.java				
		3/org.jfree.chart.ChartMouseEvent.java				
		4/org.jfree.chart.plot.AbstractPieLabelDistributor.java				
	5	1/org.jfree.chart.annotations.TextAnnotation.java	69,42	72	66,25	64,55
		2/org.jfree.chart.block.FlowArrangement.java				
		3/org.jfree.chart.ui.RectangleInsets.java				
		4/org.jfree.chart.util.DefaultShadowGenerator.java				
		Moyenne%	64,148	68,702	52,13666667	60,85
		Taux D'échec%	0		40	
GanttProject	1	1/net.sourceforge.ganttproject.calendar.CalendarActivityImpl.java	89,06	87,5	30,92	43,18
		2/net.sourceforge.ganttproject.FacadeInvalidator.java				
		3/net.sourceforge.ganttproject.gui.GanttMetalTheme.java				
		4/net.sourceforge.ganttproject.chart.ChartRendererBase .java				
	2	1/net.sourceforge.ganttproject.TreeTableCellEditorImpl.java	63,88	75	58,62	54,46
		2/net.sourceforge.ganttproject.calendar.GPCalendarBase.java				
		3/net.sourceforge.ganttproject.filter.ExtensionBasedFileFilter.java				
		4/net.sourceforge.ganttproject.action.ResourceActionSet.java				
	3	1/net.sourceforge.ganttproject.GPLoggertest.java	46,1	49,52	29,54	44,44
		2/net.sourceforge.ganttproject.calendar.GanttDaysOff.java				
		3/net.sourceforge.ganttproject.gui.options.ExportSettingsPanel.java				
		4/net.sourceforge.ganttproject.action.project.ProjectMenu.java				
	4	1/net.sourceforge.ganttproject.chart.BottomUnitLineRendererImpl.java	53,14	47,58	33,71	41,66
		2/net.sourceforge.ganttproject.calendar.AlwaysWorkingTimeCalendarImpl.java				
		3/net.sourceforge.ganttproject.document.FileDocument.java				
		4/net.sourceforge.ganttproject.gui.GanttDialogPerson.java				
	5	1/net.sourceforge.ganttproject.chart.ChartModelImpl.java	80	71,11	37,5	42,85
		2/net.sourceforge.ganttproject.document.AbstractDocument.java				
		3/net.sourceforge.ganttproject.export.ConsoleUIFacade.java				
		4/net.sourceforge.ganttproject.filter.ExtensionBasedFileFilter.java				
		Moyenne%	66,436	66,142	38,058	45,318
		Taux D'échec%	0		0	

ANNEXE II

LES RÉSULTATS DE LA TROISIÈME QUESTION EN DÉTAIL

Dans les tableaux qui suivent, nous allons présenter en détails les résultats de l'amélioration de la qualité après le refactoring 'Extract class', par et JDeodorant et BlobBreak pour chaque projet.

1. Ant Apache

- La première colonne des Tableaux II-1, II-4, II-7, II-10 et II-13 présente les 10 classes de trop grande taille choisies aléatoirement pour les projet AntAppache, GanttProject, JFreechart, Rhino et Xerces, respectivement.

Tableau-A II-1 Avant Refactoring

Classe	**CBO**	**LCOM**	**CAMC**
org.apache.tools.ant.AntClassLoader	11	1584	0,001
org.apache.tools.ant.ComponentHelper	11	788	0,1348
org.apache.tools.ant.taskdefs.optional.junit.FormatterElement	8	117	0,1056
org.apache.tools.ant.taskdefs.AbstractCvsTask	10	981	0,1111
org.apache.tools.ant.IntrospectionHelper	28	2626	0,1405
org.apache.tools.ant.taskdefs.optional.pvcs.Pvcs	10	528	0,125
org.apache.tools.ant.taskdefs.SubAnt	15	420	0,0603
org.apache.tools.ant.taskdefs.optional.ejb.WeblogicDeploymentTool	11	413	0,1296
org.apache.tools.ant.taskdefs.RecorderEntry	5	261	0,1234
org.apache.tools.ant.PropertyHelper	11	1088	0,1364
Moyenne	**12**	**880,6**	**0,10677**

- La première colonne des Tableaux II-2, II-5, II-8, II-11 et II-14 présente les 10 classes de trop grande taille choisies aléatoirement et les classes extraites par JDeodorant pour les projet AntAppache, GanttProject, JFreechart, Rhino et Xerces, respectivement.

Tableau-A II-2 Aprés Refactoring par JDeodorant

Classe	CBO	LCOM	CAMC
org.apache.tools.ant.AntClassLoader	13	1304	0,1
org.apache.tools.ant.AntClassLoaderProduct1	0	4	0,4167
org.apache.tools.ant.AntClassLoaderProduct2	1	13	0,3124
Partition	**14**	**440,3333333**	**0,276366667**
org.apache.tools.ant.ComponentHelper	14	824	0,1348
org.apache.tools.ant.ComponentHelperProduct1	2	2	0,4
org.apache.tools.ant.ComponentHelperProduct2	0	1	0,5
org.apache.tools.ant.ComponentHelperProduct3	0	214	0,1118
Partition	**16**	**260,25**	**0,28665**
org.apache.tools.ant.taskdefs.optional.junit.FormatterElement	7	26	0,1056
org.apache.tools.ant.taskdefs.optional.junit.FormatterElementProduct1	6	35	0,1296
Partition	**13**	**30,5**	**0,1176**
org.apache.tools.ant.taskdefs.AbstractCvsTask	10	926	0,1
org.apache.tools.ant.taskdefs.AbstractCvsTaskProduct1	6	42	0,1923
Partition	**16**	**484**	**0,14615**
org.apache.tools.ant.IntrospectionHelper	18	629	0,1429
org.apache.tools.ant.IntrospectionHelperProduct1	12	76	0,16
org.apache.tools.ant.IntrospectionHelperProduct2	20	621	0,1848
Partition	**50**	**442**	**0,162566667**
org.apache.tools.ant.taskdefs.optional.pvcs.Pvcs	10	528	0,125
Partition	**10**	**528**	**0,125**
org.apache.tools.ant.taskdefs.SubAnt	17	382	0,0569
org.apache.tools.ant.taskdefs.SubAntProduct1	6	27	0,1558
Partition	**23**	**204,5**	**0,10635**
org.apache.tools.ant.taskdefs.optional.ejb.WeblogicDeploymentTool	11	413	0,1296
Partition	**11**	**413**	**0,1296**
org.apache.tools.ant.taskdefs.RecorderEntry	6	216	0,1234
org.apache.tools.ant.taskdefs.RecorderEntryProduct	0	0	0,3333
Partition	**6**	**108**	**0,22835**
org.apache.tools.ant.PropertyHelper	12	864	0,1364
org.apache.tools.ant.PropertyHelperProduct1	0	10	0,3333
org.apache.tools.ant.PropertyHelperProduct2	4	24	0,2041
org.apache.tools.ant.PropertyHelperProduct3	1	10	0,3333
Partition	**17**	**227**	**0,251775**
Moyenne	**17,6**	**313,7583333**	**0,183040833**

- La première colonne des Tableaux II-3, II-6, II-9, II-12 et II-15 présente les 10 classes de trop grande taille choisies aléatoirement et les classes extraites par BlobBreak pour les projet AntAppache, GanttProject, JFreechart, Rhino et Xerces, respectivement.

Tableau-A II-3 Aprés Refactoring par BlobBreak

Classe	CBO	LCOM	CAMC
org.apache.tools.ant.AntClassLoader	11	1181	0,2162
org.apache.tools.ant.AntClassLoaderProduct1	1	120	0,1957
org.apache.tools.ant.AntClassLoaderProduct2	2	26	0,25
Partition	**14**	**442,3333333**	**0,220633333**
org.apache.tools.ant.ComponentHelper	12	489	0,1636
org.apache.tools.ant.ComponentHelperProduct1	0	0	0,5
org.apache.tools.ant.ComponentHelperProduct2	2	22	0,2286
Partition	**14**	**170,3333333**	**0,2974**
org.apache.tools.ant.taskdefs.optional.junit. FormatterElement	7	20	0,125
org.apache.tools.ant.taskdefs.optional.junit. FormatterElementProduct1	2	3	0,25
Partition	**9**	**11,5**	**0,1875**
org.apache.tools.ant.taskdefs.AbstractCvsTask	4	384	0,1347
org.apache.tools.ant.taskdefs.AbstractCvsTaskProduct1	5	29	0,225
org.apache.tools.ant.taskdefs.AbstractCvsTaskProduct2	7	41	0,3667
Partition	**16**	**151,3333333**	**0,242133333**
org.apache.tools.ant.IntrospectionHelper	24	476	0,1536
org.apache.tools.ant.IntrospectionHelperProduct1	18	400	0,2045
Partition	**42**	**438**	**0,17905**
org.apache.tools.ant.taskdefs.optional.pvcs.Pvcs	10	300	0,125
org.apache.tools.ant.taskdefs.optional.pvcs.PvcsProduct	1	28	0,8889
Partition	**11**	**328**	**0,50695**
org.apache.tools.ant.taskdefs.SubAnt	11	145	0,0724
org.apache.tools.ant.taskdefs.SubAntProduct1	5	31	0,1429
org.apache.tools.ant.taskdefs.SubAntProduct2	5	31	0,3333
Partition	**21**	**69**	**0,182866667**
org.apache.tools.ant.taskdefs.optional.ejb. WeblogicDeploymentTool	11	309	0,149
org.apache.tools.ant.taskdefs.optional.ejb. WeblogicDeploymentToolProduct	1	8	0,4
Partition	**12**	**158,5**	**0,2745**
org.apache.tools.ant.taskdefs.RecorderEntry	6	137	0,1103
org.apache.tools.ant.taskdefs.RecorderEntryProduct	1	34	0,8889
Partition	**7**	**85,5**	**0,4996**
org.apache.tools.ant.PropertyHelper	12	647	0,1952
org.apache.tools.ant.PropertyHelperProduct1	1	8	0,4
org.apache.tools.ant.PropertyHelperProduct2	2	4	0,3125
Partition	**15**	**219,6666667**	**0,302566667**
Moyenne	**16,1**	**207,4166667**	**0,28932**

2. GanttProject

Tableau-A II-4 Avant Refactoring

Classe	CBO	LCOM	CAMC
net.sourceforge.ganttproject.GanttOptions.java	23	7461	0,0796
net.sourceforge.ganttproject.GanttProject.java	107	13777	0,0184
net.sourceforge.ganttproject.task.TaskManagerImpl.java	50	892	0,0493
net.sourceforge.ganttproject.chart.ChartModelBase.java	35	1431	0,049
net.sourceforge.ganttproject.chart.SimpleRenderedImage.java	0	547	0,1143
net.sourceforge.ganttproject.document.HttpDocument.java	6	112	0,1491
net.sourceforge.ganttproject.gui.GanttTaskPropertiesBean.java	21	498	0,0982
org.ganttproject.chart.pert.ActivityOnNodePertChart.java	15	1646	0,0626
net.sourceforge.ganttproject.resource.HumanResourceManager.java	14	304	0,0848
org.ganttproject.impex.msproject.GanttMPXJOpen.java	25	41	0,2708
Moyenne	**29,6**	**2670,9**	**0,09761**

Tableau-A II-5 Aprés Refactoring par JDeodorant

Classe	**CBO**	**LCOM**	**CAMC**
net.sourceforge.ganttproject.GanttOptions.java	25	6957	0,0746
net.sourceforge.ganttproject.GanttOptionsProduct.java	0	0	0,8333
net.sourceforge.ganttproject.GanttOptionsProduct2.java	2	6	0,5
Partition	**27**	**2321**	**0,4693**
net.sourceforge.ganttproject.GanttProject.java	108	13777	0,0184
net.sourceforge.ganttproject.GanttProjectProduct.java	0	21	0,45
Partition	**108**	**6899**	**0,2342**
net.sourceforge.ganttproject.task.TaskManagerImpl.java	47	662	0,0488
net.sourceforge.ganttproject.task.TaskManagerImplProduct.java	13	0	0,1515
Partition	**60**	**331**	**0,10015**
net.sourceforge.ganttproject.chart.ChartModelBase.java	35	1431	0,049
Partition	**35**	**1431**	**0,049**
net.sourceforge.ganttproject.chart.SimpleRenderedImage.java	2	445	0,1143
net.sourceforge.ganttproject.chart.SimpleRenderedImageProduct.java	0	13	0,8571
net.sourceforge.ganttproject.chart.SimpleRenderedImageProduct2.java	0	5	0,8571
Partition	**2**	**154,3333333**	**0,6095**
net.sourceforge.ganttproject.document.HttpDocument.java	6	112	0,1491
Partition	**6**	**112**	**0,1491**
net.sourceforge.ganttproject.gui.GanttTaskPropertiesBean.java	22	459	0,0982
net.sourceforge.ganttproject.gui.GanttTaskPropertiesBeanProduct.java	0	0	0,375
net.sourceforge.ganttproject.gui.GanttTaskPropertiesBeanProduct2.java	3	0	0,25
Partition	**25**	**153**	**0,241066667**
org.ganttproject.chart.pert.ActivityOnNodePertChart.java	16	1529	0,0626
org.ganttproject.chart.pert.ActivityOnNodePertChartProduct.java	1	0	0,4
Partition	**17**	**764,5**	**0,2313**
net.sourceforge.ganttproject.resource.HumanResourceManager.java	14	211	0,0798
net.sourceforge.ganttproject.resource.HumanResourceManagerProduct.java	4	0	0,5
Partition	**18**	**105,5**	**0,2899**
org.ganttproject.impex.msproject.GanttMPXJOpen.java	5	15	0,2708
org.ganttproject.impex.msproject.GanttMPXJOpenProduct.java	24	28	0,1455
Partition	**29**	**21,5**	**0,20815**
Moyenne	**32,7**	**1229,283333**	**0,258166667**

Tableau-A II-6 Aprés Refactoring par BlobBreak

Classe	cbo	LCOM	CAMC
net.sourceforge.ganttproject.GanttOptions.java	23	4208	0,0854
net.sourceforge.ganttproject.GanttOptionsProduct.java	0	1	0,9357
net.sourceforge.ganttproject.GanttOptionsProduct2.java	2	21	0,45
Partition	**25**	**1410**	**0,490366667**
net.sourceforge.ganttproject.GanttProject.java	99	6277	0,0236
net.sourceforge.ganttproject.GanttProjectProduct.java	18	43	0,52
Partition	**117**	**3160**	**0,2718**
net.sourceforge.ganttproject.task.TaskManagerImpl.java	50	674	0,0536
net.sourceforge.ganttproject.task.TaskManagerImplProduct.java	2	9	0,1786
Partition	**52**	**341,5**	**0,1161**
net.sourceforge.ganttproject.chart.ChartModelBase.java	33	988	0,049
net.sourceforge.ganttproject.chart.ChartModelBaseProduct.java	9	64	0,2037
Partition	**21**	**526**	**0,12635**
net.sourceforge.ganttproject.chart.SimpleRenderedImage.java	1	25	0,1375
net.sourceforge.ganttproject.chart.SimpleRenderedImageProduct.java	0	28	0,952
net.sourceforge.ganttproject.chart.SimpleRenderedImageProduct2.java	1	155	0,9474
Partition	**2**	**69,33333333**	**0,678966667**
net.sourceforge.ganttproject.document.HttpDocument.java	6	62	0,1569
net.sourceforge.ganttproject.document.HttpDocumentProduct.java	4	1	0,9142
Partition	**10**	**31,5**	**0,53555**
net.sourceforge.ganttproject.gui.GanttTaskPropertiesBean.java	21	239	0,0889
net.sourceforge.ganttproject.gui.GanttTaskPropertiesBeanProduct.java	0	62	0,3333
net.sourceforge.ganttproject.gui.GanttTaskPropertiesBeanProduct2.java	4	10	0,622
Partition	**25**	**103,6666667**	**0,348066667**
org.ganttproject.chart.pert.ActivityOnNodePertChart.java	15	1328	0,0581
org.ganttproject.chart.pert.ActivityOnNodePertChartProduct.java	4	15	0,5625
Partition	**19**	**671,5**	**0,3103**
net.sourceforge.ganttproject.resource.HumanResourceManager.java	15	191	0,0889
net.sourceforge.ganttproject.resource.HumanResourceManagerProduct.java	4	9	0,7286
Partition	**19**	**100**	**0,40875**
org.ganttproject.impex.msproject.GanttMPXJOpen.java	6	13	0,2708
org.ganttproject.impex.msproject.GanttMPXJOpenProduct.java	23	8	0,8
Partition	**29**	**10,5**	**0,5354**
Moyenne	**31,9**	**642,4**	**0,382165**

3. Jfreechart

Tableau-A II-7 Avant Refactoring

Classe	**CBO**	**LCOM**	**CAMC**
org.jfree.chart.block.LabelBlock	13	113	0,1318
org.jfree.chart.editor.DefaultAxisEditor	11	107	0,1111
org.jfree.chart.title.LegendGraphic	10	447	0,1019
org.jfree.chart.ChartPanel	21	5762	0,043
org.jfree.chart.axis.SymbolAxis	12	107	0,2201
org.jfree.data.gantt.Task	4	45	0,1407
org.jfree.data.general.Series	3	139	0,1256
org.jfree.data.general.WaferMapDataset	2	121	0,2281
org.jfree.chart.entity.ChartEntity	5	76	0,1875
org.jfree.data.time.TimeSeries	7	949	0,0932
Moyenne	**8,8**	**786,6**	**0,1383**

Tableau-A II-8 Aprés Refactoring avec JDeodorant

Classe	**CBO**	**LCOM**	**CAMC**
org.jfree.chart.block.LabelBlock	13	101	0,1318
org.jfree.chart.block.LabelBlockProduct	3	24	0,1833
Partition	**16**	**62,5**	**0,15755**
org.jfree.chart.editor.DefaultAxisEditor	12	96	0,1111
org.jfree.chart.editor.DefaultAxisEditorProduct	2	3	0,2778
org.jfree.chart.editor.DefaultAxisEditorProduct2	2	3	0,2778
Partition	**16**	**34**	**0,222233333**
org.jfree.chart.title.LegendGraphic	10	411	0,1019
org.jfree.chart.title.LegendGraphicProduct	4	37	0,303
Partition	**14**	**224**	**0,20245**
org.jfree.chart.ChartPanel	21	5762	0,043
Partition	**21**	**5762**	**0,043**
org.jfree.chart.axis.SymbolAxis	13	95	0,2136
org.jfree.chart.axis.SymbolAxisProduct	1	3	0,25
Partition	**14**	**49**	**0,2318**
org.jfree.data.gantt.Task	5	33	0,1407
org.jfree.data.gantt.TaskProduct	2	0	0,25
Partition	**7**	**16,5**	**0,19535**
org.jfree.data.general.Series	7	115	0,1256
org.jfree.data.general.SeriesProduct	1	1	0,25
org.jfree.data.general.SeriesProduct2	3	3	0,2083
org.jfree.data.general.SeriesProduct3	4	4	0,25
org.jfree.data.general.SeriesProduct4	1	1	0,25
Partition	**16**	**24,8**	**0,21678**
org.jfree.data.general.WaferMapDataset	5	48	0,1929
org.jfree.data.general.WaferMapDatasetProduct	2	3	0,2
org.jfree.data.general.WaferMapDatasetProduct2	2	12	0,2222
Partition	**9**	**21**	**0,205033333**
org.jfree.chart.entity.ChartEntity	5	27	0,1875
org.jfree.chart.entity.ChartEntityProduct	2	10	0,35
Partition	**7**	**18,5**	**0,26875**
org.jfree.data.time.TimeSeries	8	931	0,0932
org.jfree.data.time.TimeSeries	2	16	0,375
Partition	**10**	**473,5**	**0,2341**
Moyenne	**13**	**668,58**	**0,197704667**

Tableau-A II-9 Aprés Refactoring avec BlobBreak

Classe	CBO	LCOM	CAMC
org.jfree.chart.block.LabelBlock	12	52	0,163
org.jfree.chart.block.LabelBlockProduct	3	0	0,2333
Partition	**15**	**26**	**0,19815**
org.jfree.chart.editor.DefaultAxisEditor	10	46	0,1571
org.jfree.chart.editor.DefaultAxisEditorProduct	2	22	0,25
org.jfree.chart.editor.DefaultAxisEditorProduct2	3	15	0,3333
Partition	**15**	**27,66666667**	**0,2468**
org.jfree.chart.title.LegendGraphic	9	103	0,14
org.jfree.chart.title.LegendGraphicProduct	2	6	0,4
org.jfree.chart.title.LegendGraphicProduct2	1	66	0,2308
Partition	**12**	**58,33333333**	**0,256933333**
org.jfree.chart.ChartPanel	22	4569	0,0432
org.jfree.chart.ChartPanelProduct	0	110	0,3
Partition	**22**	**2339,5**	**0,1716**
org.jfree.chart.axis.SymbolAxis	11	30	0,28
org.jfree.chart.axis.SymbolAxisProduct	3	42	0,2468
Partition	**14**	**36**	**0,2634**
org.jfree.data.gantt.Task	5	6	0,275
org.jfree.data.gantt.TaskProduct	1	4	0,375
org.jfree.data.gantt.TaskProduct2	2	0	0,3333
Partition	**8**	**3,333333333**	**0,327766667**
org.jfree.data.general.Series	2	21	0,24
org.jfree.data.general.SeriesProduct	1	4	0,2813
org.jfree.data.general.SeriesProduct2	3	9	0,2381
Partition	**6**	**11,33333333**	**0,253133333**
org.jfree.data.general.WaferMapDataset	2	13	0,33
org.jfree.data.general.WaferMapDatasetProduct	2	30	0,2778
org.jfree.data.general.WaferMapDatasetProduct2	1	3	0,4444
Partition	**5**	**15,33333333**	**0,350733333**
org.jfree.chart.entity.ChartEntity	6	46	0,2024
org.jfree.chart.entity.ChartEntityProduct	2	11	0,3889
Partition	**8**	**28,5**	**0,29565**
org.jfree.data.time.TimeSeries	7	482	0,1083
org.jfree.data.time.TimeSeries	3	118	0,1719
Partition	**10**	**300**	**0,1401**
Moyenne	**11,5**	**284,6**	**0,250426667**

4. Rhino

Tableau-A II-10 Avant Refactoring

Classe	**CBO**	**LCOM**	**CAMC**
org.mozilla.javascript.Node	11	1598	0,1214
org.mozilla.classfile.ClassFileWriter	16	3741	0,1019
org.mozilla.javascript.IRFactory	66	760	0,1111
org.mozilla.javascript.JavaAdapter	26	561	0,1474
org.mozilla.javascript.NativeArray	16	2624	0,1167
org.mozilla.javascript.Context	42	7628	0,0522
org.mozilla.javascript.TokenStream	5	1005	0,1607
org.mozilla.javascript.Parser	75	4704	0,1127
org.mozilla.javascript.ScriptableObject	28	6919	0,1535
org.mozilla.javascript.ast.Jump	4	4	0,2235
Moyenne	**28,9**	**2954,4**	**0,13011**

Tableau-A II-11 Aprés Refactoring avec JDeodorant

Classe	**CBO**	**LCOM**	**CAMC**
org.mozilla.javascript.Node	12	1407	0,1214
org.mozilla.javascript.NodeProduct	1	3	0,4444
org.mozilla.javascript.NodeProduct2	3	0	0,2
Partition	**16**	**470**	**0,255266667**
org.mozilla.classfile.ClassFileWriter	18	3783	0,0956
org.mozilla.classfile.ClassFileWriterProduct	2	7	0,3333
org.mozilla.classfile.ClassFileWriterProduct2	1	0	0,8571
Partition	**21**	**1263,333333**	**0,428666667**
org.mozilla.javascript.IRFactory	67	310	0,1065
org.mozilla.javascript.IRFactoryProduct	4	8	0,375
Partition	**71**	**159**	**0,24075**
org.mozilla.javascript.JavaAdapter	26	561	0,1474
Partition	**26**	**561**	**0,1474**
org.mozilla.javascript.NativeArray	16	2624	0,1167
Partition	**16**	**2624**	**0,1167**
org.mozilla.javascript.Context	43	7542	0,0518
org.mozilla.javascript.ContextProduct	1	15	0,3636
org.mozilla.javascript.ContextProduct2	2	6	0,4667
org.mozilla.javascript.ContextProduct3	6	104	0,1711
org.mozilla.javascript.ContextProduct4	5	1	0,2063
Partition	**57**	**1533,6**	**0,2519**
org.mozilla.javascript.TokenStream	5	1005	0,1607
Partition	**5**	**1005**	**0,1607**
org.mozilla.javascript.Parser	79	2542	0,1044
org.mozilla.javascript.ParserProduct	0	4	0,5
org.mozilla.javascript.ParserProduct2	1	0	0,5714
org.mozilla.javascript.ParserProduct3	5	6	0,3333
org.mozilla.javascript.ParserProduct4	5	9	0,1818
Partition	**90**	**512,2**	**0,33818**
org.mozilla.javascript.ScriptableObject	28	6919	0,1535
Partition	**28**	**6919**	**0,1535**
org.mozilla.javascript.ast.Jump	5	4	0,2235
org.mozilla.javascript.ast.JumpProduct	2	13	0,6061
Partition	**7**	**8,5**	**0,4148**
Moyenne	**33,7**	**1505,563333**	**0,250786333**

Tableau-A II-12 Aprés Refactoring avec BlobBreak

Classe	**CBO**	**LCOM**	**CAMC**
org.mozilla.javascript.Node	12	855	0,142
org.mozilla.javascript.NodeProduct	2	0	1
org.mozilla.javascript.NodeProduct2	2	37	0,4167
Partition	**16**	**297,3333333**	**0,519566667**
org.mozilla.classfile.ClassFileWriter	15	2000	0,1405
org.mozilla.classfile.ClassFileWriterProduct	11	935	0,1827
Partition	**26**	**1467,5**	**0,1616**
org.mozilla.javascript.IRFactory	66	219	0,1605
org.mozilla.javascript.IRFactoryProduct	9	123	0,1929
org.mozilla.javascript.IRFactoryProduct2	2	0	0,9
Partition	**77**	**114**	**0,4178**
org.mozilla.javascript.JavaAdapter	28	217	0,1536
org.mozilla.javascript.JavaAdapterProduct	2	71	0,325
Partition	**30**	**288**	**0,4786**
org.mozilla.javascript.NativeArray	16	2329	0,1185
org.mozilla.javascript.NativeArrayProduct	6	26	0,3
Partition	**22**	**1177,5**	**0,20925**
org.mozilla.javascript.Context	38	3338	0,0722
org.mozilla.javascript.ContextProduct	2	376	0,1461
org.mozilla.javascript.ContextProduct2	6	76	0,1635
org.mozilla.javascript.ContextProduct3	7	7	0,8124
Partition	**53**	**949,25**	**0,29855**
org.mozilla.javascript.TokenStream	5	666	0,2202
org.mozilla.javascript.TokenStreamProduct	1	19	0,2381
org.mozilla.javascript.TokenStreamProduct2	1	66	0,2727
Partition	**7**	**250,3333333**	**0,243666667**
org.mozilla.javascript.Parser	52	1389	0,0821
org.mozilla.javascript.ParserProduct	7	64	0,375
org.mozilla.javascript.ParserProduct2	1	195	0,4
org.mozilla.javascript.ParserProduct3	25	204	0,2778
org.mozilla.javascript.ParserProduct4	30	33	0,3704
Partition	**115**	**377**	**0,30106**
org.mozilla.javascript.ScriptableObject	29	6919	0,1535
org.mozilla.javascript.ScriptableObjectProduct	15	300	0,3156
Partition	**44**	**3609,5**	**0,23455**
org.mozilla.javascript.ast.Jump	4	25	0,2364
org.mozilla.javascript.ast.JumpProduct	2	6	0,4444
Partition	**6**	**15,5**	**0,3404**
Moyenne	**39,6**	**854,5916667**	**0,320504333**

5. Xerces

Tableau-A II-13 Avant Refactoring

Classe	**CBO**	**LCOM**	**CAMC**
org.apache.xerces.parsers.AbstractDOMParser	44	89	0,1085
org.apache.xerces.impl.xpath.regex.RegularExpression	17	1082	0,2554
org.apache.xerces.impl.xpath.regex.RegexParser	9	706	0,1641
org.apache.xerces.impl.dv.xs.AbstractDateTimeDV	6	374	0,1697
org.apache.xerces.impl.dtd.DTDGrammar	36	4055	0,0869
org.apache.xerces.impl.XMLErrorReporter	9	134	0,1483
org.apache.xerces.util.DOMUtil	10	1081	0,3313
org.apache.xerces.dom.DeferredDocumentImpl	26	2258	0,1961
org.apache.xerces.xinclude.XIncludeHandler	41	4760	0,0708
org.apache.xerces.impl.XMLDTDScannerImpl	18	306	0,0786
Moyenne	**21,6**	**1484,5**	**0,16097**

Tableau-A II-14 Aprés Refactoring avec JDeodorant

Classe	CBO	LCOM	CAMC
org.apache.xerces.impl.xpath.regex.RegexParser	44	89	0,1085
Partition	**44**	**89**	**0,1085**
org.apache.xerces.impl.xpath.regex.RegularExpression	17	1059	0,2983
org.apache.xerces.impl.xpath.regex.RegularExpressionProduct	5	0	0,75
Partition	**22**	**529,5**	**0,52415**
org.apache.xerces.impl.xpath.regex.RegexParser	11	494	0,1641
org.apache.xerces.impl.xpath.regex.RegexParserProduct	1	4	0,4167
org.apache.xerces.impl.xpath.regex.RegexParserProduct2	1	1	0,3333
Partition	**13**	**166,3333333**	**0,3047**
org.apache.xerces.impl.dv.xs.AbstractDateTimeDV	7	344	0,1697
org.apache.xerces.impl.dv.xs.AbstractDateTimeDVProduct	0	6	0,75
Partition	**7**	**175**	**0,45985**
org.apache.xerces.impl.dtd.DTDGrammar	39	4017	0,0816
org.apache.xerces.impl.dtd.DTDGrammarProduct	5	24	0,2154
org.apache.xerces.impl.dtd.DTDGrammarProduct2	7	3	0,3667
org.apache.xerces.impl.dtd.DTDGrammarProduct3	2	0	0,4
org.apache.xerces.impl.dtd.DTDGrammarProduct4	5	0	0,1964
Partition	**58**	**808,8**	**0,25202**
org.apache.xerces.impl.XMLErrorReporter	10	126	0,1483
org.apache.xerces.impl.XMLErrorReporterProduct	1	0	0,4667
Partition	**11**	**63**	**0,3075**
org.apache.xerces.util.DOMUtil	10	1081	0,3313
Partition	**10**	**1081**	**0,3313**
org.apache.xerces.dom.DeferredDocumentImpl	29	2397	0,1968
org.apache.xerces.dom.DeferredDocumentImplProduct	3	9	0,3333
Partition	**32**	**1203**	**0,26505**
org.apache.xerces.xinclude.XIncludeHandler	43	4499	0,0708
org.apache.xerces.xinclude.XIncludeHandlerProduct	3	10	0,48
org.apache.xerces.xinclude.XIncludeHandlerProduct2	1	0	0,625
Partition	**47**	**1503**	**0,391933333**
org.apache.xerces.impl.XMLDTDScannerImpl	18	216	0,0778
org.apache.xerces.impl.XMLDTDScannerImplProduct2	3	0	0,8
org.apache.xerces.impl.XMLDTDScannerImplProduct	1	0	0,5833
Partition	**22**	**72**	**0,487033333**
Moyenne	**26,6**	**569,0633333**	**0,343203667**

Tableau-A II-15 Aprés Refactoring avec BlobBreak

Classe	CBO	LCOM	CAMC
org.apache.xerces.parsers.AbstractDOMParser	43	103	0,1106
org.apache.xerces.parsers.AbstractDOMParserpRODUCT	8	6	0,4167
Partition	**51**	**54,5**	**0,26365**
org.apache.xerces.impl.xpath.regex.RegularExpression	14	608	0,2552
org.apache.xerces.impl.xpath.regex.RegularExpressionProduct	7	62	0,2778
Partition	**21**	**335**	**0,2665**
org.apache.xerces.impl.xpath.regex.RegexParser	10	569	0,1852
org.apache.xerces.impl.xpath.regex.RegexParserProduct	1	4	0,85
Partition	**11**	**286,5**	**0,5176**
org.apache.xerces.impl.dv.xs.AbstractDateTimeDV	6	203	0,1856
org.apache.xerces.impl.dv.xs.AbstractDateTimeDVProduct	1	150	0,3333
Partition	**7**	**176,5**	**0,25945**
org.apache.xerces.impl.dtd.DTDGrammar	33	1144	0,1008
org.apache.xerces.impl.dtd.DTDGrammarElementProduct	9	91	0,2262
org.apache.xerces.impl.dtd.DTDGrammarContentProduct2	6	53	0,3167
org.apache.xerces.impl.dtd.DTDGrammarAttributeProduct3	9	81	0,2143
org.apache.xerces.impl.dtd.DTDGrammarEntityProduct4	4	19	0,4286
org.apache.xerces.impl.dtd.DTDGrammarNotationProduct5	5	13	0,3056
Partition	**66**	**233,5**	**0,265366667**
org.apache.xerces.impl.XMLErrorReporter	10	55	0,286
org.apache.xerces.impl.XMLErrorReporterProduct	1	4	0,375
Partition	**11**	**29,5**	**0,3305**
org.apache.xerces.util.DOMUtil	11	820	0,3862
org.apache.xerces.util.DOMUtilProduct	3	21	0,6667
Partition	**14**	**420,5**	**0,52645**
org.apache.xerces.dom.DeferredDocumentImpl	26	1525	0,2385
org.apache.xerces.dom.DeferredDocumentImplProduct	4	15	0,3167
org.apache.xerces.dom.DeferredDocumentImplChunkProduct	4	52	0,2667
Partition	**34**	**530,6666667**	**0,273966667**
org.apache.xerces.xinclude.XIncludeHandler	42	3816	0,0703
org.apache.xerces.xinclude.XIncludeHandlerErrorProduct	1	15	0,6
org.apache.xerces.xinclude.XIncludeHandlerProduct2	2	10	0,8
Partition	**45**	**1280,333333**	**0,4901**
org.apache.xerces.impl.XMLDTDScannerImpl	16	115	0,0924
org.apache.xerces.impl.XMLDTDScannerImplIPEProduct	3	0	1
org.apache.xerces.impl.XMLDTDScannerImplContentProduct	2	0	0,9951
org.apache.xerces.impl.XMLDTDScannerImplProduct	3	120	0,2115
Partition	**24**	**58,75**	**0,57475**
Moyenne	**28,4**	**340,575**	**0,376833333**

RÉFÉRENCES

Alba, E. & Chicano, F. (2005). Management of software projects with GAs. *Proceedings of the 6th metaheuristics international conference (MIC'05)*, pp. 13–18.

Alikacem, E. H. & Sahraoui, H. A. (2009). A metric extraction framework based on a high-level description language. *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 159–167.

Baudry, B., Le Traon, Y. & Sunyé, G. (2004). Improving the testability of UML class diagrams. *First International Workshop onTestability Assessment, 2004. IWoTA 2004. Proceedings.*, pp. 70–80.

Bavota, G., De Lucia, A., Marcus, A. & Oliveto, R. (2010a). A two-step technique for extract class refactoring. *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 151–154.

Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G. & Gueheneuc, Y.-G. (2010b). Playing with refactoring : Identifying extract class opportunities through game theory. *2010 IEEE International Conference on Software Maintenance*, pp. 1–5.

Bavota, G., De Lucia, A., Marcus, A., Oliveto, R. & Palomba, F. (2012). Supporting extract class refactoring in eclipse : The aries project. *Proceedings of the 34th International Conference on Software Engineering*, pp. 1419–1422.

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M. & Devanbu, P. (2009). The promises and perils of mining git. *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 1–10.

Boukharata, S., Ouni, A., Kessentini, M., Bouktif, S. & Wang, H. (2019). Improving web service interfaces modularity using multi-objective optimization. *Automated Software Engineering*, 26(2), 275–312.

Brown, W. H., Malveau, R. C., McCormick, H. W. & Mowbray, T. J. (1998). *AntiPatterns : refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Chatzigeorgiou, A. & Manakos, A. (2010). Investigating the evolution of bad smells in object-oriented code. *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pp. 106–115.

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6), 476–493.

Coello, C. A. C., Lamont, G. B., Van Veldhuizen, D. A. et al. (2007). *Evolutionary algorithms for solving multi-objective problems*. Springer.

Das, I. & Dennis, J. E. (1998). Normal-boundary intersection : A new method for generating the Pareto surface in nonlinear multicriteria optimization problems. *SIAM journal on optimization*, 8(3), 631–657.

De Lucia, A., Oliveto, R. & Vorraro, L. (2008). Using structural and semantic metrics to improve class cohesion. *2008 IEEE International Conference on Software Maintenance*, pp. 27–36.

de Oliveira, M. C., Freitas, D., Bonifácio, R., Pinto, G. & Lo, D. (2019). Finding needles in a haystack : Leveraging co-change dependencies to recommend refactorings. *Journal of Systems and Software*, 158, 110420.

Deb, K. & Jain, H. (2013). An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I : solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4), 577–601.

Deb, K., Pratap, A., Agarwal, S. & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm : NSGA-II. *IEEE transactions on evolutionary computation*, 6(2), 182–197.

Ekman, T., Schäfer, M. & Verbaere, M. (2008). Refactoring is not (yet) about transformation. *Proceedings of the 2nd Workshop on Refactoring Tools*, pp. 5.

Feng, T., Zhang, J., Wang, H. & Wang, X. (2004). Software design improvement through anti-patterns identification. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 524.

Fokaefs, M., Tsantalis, N., Stroulia, E. & Chatzigeorgiou, A. (2011). JDeodorant : identification and application of extract class refactorings. *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1037–1039.

Fokaefs, M., Tsantalis, N., Stroulia, E. & Chatzigeorgiou, A. (2012). Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10), 2241–2260.

Fowler, M. (2018). *Refactoring : improving the design of existing code*. Addison-Wesley Professional.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, d. (1999). *Refactoring : Improving the Design of Existing Code*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Ganea, G., Verebi, I. & Marinescu, R. (2017). Continuous quality assessment with inCode. *Science of Computer Programming*, 134, 19–36.

Hadka, D. & Reed, P. (2013). Borg : An auto-adaptive many-objective evolutionary computing framework. *Evolutionary computation*, 21(2), 231–259.

Harman, M. & Jones, B. F. (2001). Search-based software engineering. *Information and software Technology*, 43(14), 833–839.

Harman, M. & Tratt, L. (2007). Pareto optimal search based refactoring at the design level. *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1106–1113.

Harman, M., Mansouri, S. A. & Zhang, Y. (2012). Search-based software engineering : Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1), 11.

Holland, J. H. & Goldberg, D. E. (1988). Genetic algorithms and machine learning. *Machine learning*, 3(2), 95–99.

Joshi, P. & Joshi, R. K. (2009). Concept analysis for class cohesion. *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 237–240.

Kessentini, M., Mahaouachi, R. & Ghedira, K. (2013). What you like in design use to correct bad-smells. *Software Quality Journal*, 21(4), 551–571.

Kim, D. K. (2017). Finding Bad Code Smells with Neural Network Models. *International Journal of Electrical & Computer Engineering (2088-8708)*, 7(6).

Kruskal, W. H. et al. (1952). A nonparametric test for the several sample problem. *The Annals of Mathematical Statistics*, 23(4), 525–540.

Lachance, É. (2014). Résolution de conflits et séquençage d'avions par algorithmes évolutionnaires multiobjectifs.

Le Goues, C., Nguyen, T., Forrest, S. & Weimer, W. (2011). Genprog : A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1), 54–72.

Liu, H., Yang, L., Niu, Z., Ma, Z. & Shao, W. (2009). Facilitating software refactoring with appropriate resolution order of bad smells. *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 265–268.

Mann, H. B. & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.

Marinescu, R. (2004). Detection strategies : Metrics-based rules for detecting design flaws. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350–359.

Martin, R. C. (2002). *Agile software development : principles, patterns, and practices*. Prentice Hall.

Martin, R. C. & Martin, M. (2006). *Agile principles, patterns, and practices in C# (Robert C. Martin)*. Prentice Hall PTR.

McMinn, P. (2004). Search-based software test data generation : a survey. *Software testing, Verification and reliability*, 14(2), 105–156.

Mealy, E., Carrington, D., Strooper, P. & Wyeth, P. (2007). Improving usability of software refactoring tools. *2007 Australian Software Engineering Conference (ASWEC'07)*, pp. 307–318.

Mkaouer, W., Kessentini, M., Shaout, A., Koligheu, P., Bechikh, S., Deb, K. & Ouni, A. (2015). Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3), 17.

Moha, N., Gueheneuc, Y.-G., Duchien, L. & Le Meur, A.-F. (2009). Decor : A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 20–36.

Murphy-Hill, E., Black, A. P., Dig, D. & Parnin, C. (2008). Gathering refactoring data : a comparison of four methods. *Proceedings of the 2nd Workshop on Refactoring Tools*, pp. 7.

Opdyke, W. F. (1992). Refactoring object-oriented frameworks.

Ouni, A., Kessentini, M., Sahraoui, H. & Hamdi, M. S. (2012). Search-based refactoring : Towards semantics preservation. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 347–356.

Ouni, A., Kessentini, M. & Sahraoui, H. (2013a). Search-based refactoring using recorded code changes. *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 221–230.

Ouni, A., Kessentini, M., Sahraoui, H. & Boukadoum, M. (2013b). Maintainability defects detection and correction : a multi-objective approach. *Automated Software Engineering*, 20(1), 47–79.

Ouni, A., Kessentini, M., Inoue, K. & Cinnéide, M. O. (2015). Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, 10(4), 603–617.

Ouni, A., Salem, Z., Inoue, K. & Soui, M. (2016). SIM : an automated approach to improve web service interface modularization. *2016 IEEE International Conference on Web Services (ICWS)*, pp. 91–98.

Ouni, A., Wang, H., Kessentini, M., Bouktif, S. & Inoue, K. (2018). A hybrid approach for improving the design quality of web service interfaces. *ACM Transactions on Internet Technology (TOIT)*, 19(1), 4.

Pareto, V. (1964). *Cours d'économie politique*. Librairie Droz.

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

Srinivas, N. & Deb, K. (1994). Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2(3), 221–248.

Travassos, G., Shull, F., Fredericks, M. & Basili, V. R. (1999). Detecting defects in object-oriented designs : using reading techniques to increase software quality. *ACM Sigplan Notices*, 34(10), 47–56.

Wang, H., Ouni, A., Kessentini, M., Maxim, B. & Grosky, W. I. (2016). Identification of web service refactoring opportunities as a multi-objective problem. *2016 IEEE International Conference on Web Services (ICWS)*, pp. 586–593.

Wang, H., Kessentini, M. & Ouni, A. (2017). Interactive refactoring of web service interfaces using computational search. *IEEE Transactions on Services Computing*.

Zhang, M., Baddoo, N., Wernick, P. & Hall, T. (2011). Prioritising refactoring using code bad smells. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 458–464.

Zitzler, E., Laumanns, M. & Thiele, L. (2001). SPEA2 : Improving the strength Pareto evolutionary algorithm. *TIK-report*, 103.