2M11.3275.1

Université de Montréal

Estimation de l'impact du changement dans les programmes à Objets

Par
Laila Cheikhi

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès Sciences (M.Sc)
en Informatique

Octobre, 2004

Copyright, Laila Cheikhi, 2004

QA
76
U54
2005
v.005

Université de Montréal

Direction des bibliothèques

AVIS

L'auteur a autorisé l'Université de Montréal à reproduire et diffuser, en totalité ou en partie, par quelque moyen que ce soit et sur quelque support que ce soit, et exclusivement à des fins non lucratives d'enseignement et de recherche, des copies de ce mémoire ou de cette thèse.

L'auteur et les coauteurs le cas échéant conservent la propriété du droit d'auteur et des droits moraux qui protègent ce document. Ni la thèse ou le mémoire, ni des extraits substantiels de ce document, ne doivent être imprimés ou autrement reproduits sans l'autorisation de l'auteur.

Afin de se conformer à la Loi canadienne sur la protection des renseignements personnels, quelques formulaires secondaires, coordonnées ou signatures intégrées au texte ont pu être enlevés de ce document. Bien que cela ait pu affecter la pagination, il n'y a aucun contenu manquant.

NOTICE

The author of this thesis or dissertation has granted a nonexclusive license allowing Université de Montréal to reproduce and publish the document, in part or in whole, and in any format, solely for noncommercial educational and research purposes.

The author and co-authors if applicable retain copyright ownership and moral rights in this document. Neither the whole thesis or dissertation, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms, contact information or signatures may have been removed from the document. While this may affect the document page count, it does not represent any loss of content from the document.

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Estimation de l'impact du changement dans les programmes à Objets

Présenté par :
Laila Cheikhi

a été évalué par un jury composé des personnes suivantes :

Aboulhamid El Mostapha
président-rapporteur

Sahraoui Houari
directeur de recherche

Lounis Hakim
codirecteur

Yann-Gaël Guéhéneuc
membre du jury

Mémoire accepté :
20 Décembre 2004

Sommaire

La technologie orientée objet produit des systèmes logiciels structurés et faciles à maintenir. Après livraison, les systèmes entrent en phase de maintenance au cours de laquelle des requêtes de modifications sont formulées par les utilisateurs et l'analyse et la détermination de l'impact des changements sont effectuées par l'équipe de maintenance. Une fois le code modifié et les tests réalisés, une nouvelle version du système est livrée. Aujourd'hui, la nécessité de diminuer le coût de la maintenance est devenue la tâche la plus lourde aussi bien pour les gestionnaires que pour les personnes chargées de la maintenance. Cela est dû au besoin de garder les systèmes toujours opérationnels et à jour. L'analyse d'impact du changement est l'une des techniques de maintenance permettant d'identifier la partie ou les parties du système affectées par le changement et, ainsi, fournir une estimation des ressources nécessaires pour son accomplissement.

Dans ce mémoire, nous proposons une méthodologie pour estimer l'impact du changement des systèmes orientés objets. Elle se base sur une approche d'analyse d'impact du changement que nous avons établie et sur des métriques de conception orientée objet. Nous proposons deux hypothèses qui lient les propriétés de conception orientée objet, l'héritage et le couplage, avec l'impact du changement. Puis, nous utilisons quelques algorithmes d'apprentissage pour produire nos modèles prédictifs. Dans le cadre de notre étude empirique, nous avons sélectionné un ensemble de types de changements et utilisé le modèle d'impact du changement proposé par l'approche. L'expérience est faite sur deux systèmes industriels, en collaboration avec le CRIM. Les métriques ont été calculées via deux outils : BOAP, une boîte à outils pour l'analyse de programmes, et un autre outil implémenté sous Eclipse. Enfin, nous avons évalué les résultats obtenus. Il en ressort que les métriques de conception orientée objet relatives à l'héritage et au couplage sont de bons indicateurs de l'impact du changement.

Mots Clés : *Paradigme orienté objet, Maintenance de programmes orientés objets, Analyse d'impact, Processus de changement, Métriques orientées objet, Algorithmes d'apprentissage.*

Abstract

Object oriented technology produces structured software systems, which are easy to maintain. After delivery, a system enters maintenance during which requests of modifications are formulated by the users and analysis and determination of change impacts are carried out by the maintenance team. Once the code is modified and the tests realized, a new version of the system is delivered. Today, the need for reducing the cost of maintenance is increasingly challenging managers as well as engineers in charge of maintenance. In fact, the need of keeping always systems operational and up to date is costly. Change impact analysis is one of the maintenance techniques that allows engineers and managers to identify the parts of a system affected by the change and thus to provide an estimate of the resources necessary for its achievement.

In this dissertation, we propose a methodology for estimating change impact of object-oriented systems. It is based on an approach of change impact analysis, which we elaborated using object-oriented design metrics. We propose two assumptions for binding the properties of object-oriented design, inheritance and coupling, with change impact. Then we use machine learning algorithms to produce our predictive models. In our empirical study, we selected a set of types of changes and used the model of change impact suggested by the approach. An experiment was carried on two industrial systems, in collaboration with the Computer Research Institute of Montreal (CRIM). The metrics were calculated using two tools: BOAP, a toolbox for the analysis of programs, and another tool that we implemented with Eclipse. Finally, we evaluate the methodology and analyze the results. The results reveal that object-oriented design metrics related to inheritance and coupling are good indicators of change impact.

***Keywords**: Object oriented paradigm, Object oriented maintenance, Impact analysis, Process of change, Object oriented metrics, Machine learning algorithms.*

Table des Matières

SOMMAIRE .... **I**

ABSTRACT .... **II**

TABLE DES MATIÈRES .... **III**

LISTE DES TABLEAUX .... **V**

LISTE DES FIGURES .... **VI**

LISTE DES SIGLES ET ABRÉVIATIONS .... **VII**

REMERCIEMENTS .... **VIII**

CHAPITRE 1 INTRODUCTION **1**

1.1 CONTEXTE 1
1.1.1 Maintenance des systèmes *3*
1.1.2 Techniques de maintenance *3*
1.1.3 Modèles de maintenance *4*
1.2 PROBLÉMATIQUE ET MOTIVATIONS 5
1.3 OBJECTIFS 7
1.4 ORGANISATION DU MÉMOIRE 8

CHAPITRE 2 ÉTAT DE L'ART **9**

2.1 NOTIONS DE BASE 9
2.1.1 Concepts de l'approche orientée objet *9*
2.1.2 Notion de dépendance *10*
2.1.3 Notion d'impact *13*
2.1.4 Propagation d'impact du changement *14*
2.1.5 Tests de régression *16*
2.1.6 Conclusion *17*
2.2 ANALYSE D'IMPACT DU CHANGEMENT 18
2.2.1 Processus du changement *19*
2.2.2 Analyse d'impact *22*
2.2.3 Avantages de l'analyse d'impact *23*
2.2.4 Travaux de recherche sur l'analyse d'impact des systèmes OO *24*
2.2.5 Démarches d'analyse d'impact *28*
2.2.6 Difficultés de l'analyse d'impact *29*
2.2.7 Conclusion *29*
2.3 MÉTRIQUES ORIENTÉES OBJETS 30
2.3.1 Quelques métriques de conception orientée objet *31*
2.3.2 Les métriques orientées objets et l'analyse d'impact du changement *33*
2.3.3 Conclusion *36*

CHAPITRE 3 APPROCHE DE L'ANALYSE D'IMPACT **37**

3.1 CLASSIFICATION DES CHANGEMENTS 38
3.1.1 Questionnaire des types de changements *38*
3.1.2 Analyse des résultats *40*

3.1.3 Conclusion ... *44*
3.2 DÉFINITION DU MODÈLE D'IMPACT DE CHANGEMENT ... 45
3.2.1 Nouveaux concepts/définitions ... *45*
3.2.2 Modèles d'impact du changement ... *47*
3.3 HYPOTHÈSES SUR L'IMPACT DU CHANGEMENT ... 49
3.3.1 Héritage et impact du changement ... *50*
3.3.2 Couplage et impact du changement ... *50*
3.4 EXTRACTION DES MÉTRIQUES ... 51
3.5 ESTIMATION DE L'IMPACT DE CHANGEMENT PAR LES MODÈLES PRÉDICTIFS ... 52
3.6 CONCLUSION ... 53

CHAPITRE 4 EXTRACTION DES MÉTRIQUES ... 54

4.1 EXTRACTION DES MÉTRIQUES AVEC BOAP ... 54
4.1.1 Analyse du module de métriques de BOAP ... *55*
4.1.2 Implémentation de l'extraction des métriques ... *62*
4.1.3 Conclusion ... *66*
4.2 EXTRACTION DES MÉTRIQUES AVEC ECLIPSE ... 67
4.2.1 Processus d'analyse du code source ... *67*
4.2.2 Travail effectué sous Eclipse ... *69*
4.3 CONCLUSION ... 73

CHAPITRE 5 EXPÉRIENCE ET RÉSULTATS ... 74

5.1 DONNÉES DE L'ÉTUDE ... 74
5.1.1 Systèmes étudiés ... *74*
5.1.2 Liste des métriques ... *75*
5.1.3 Liste des changements ... *76*
5.1.4 Questionnaire du calcul d'impact ... *77*
5.2 RÉSULTATS ... 79
5.2.1 Calcul des métriques ... *79*
5.2.2 Calcul d'impact ... *81*
5.3 CONSTRUCTION DES MODÈLES D'ÉVALUATION DE L'IMPACT DU CHANGEMENT ... 83
5.4 ANALYSE ET INTERPRÉTATIONS ... 85
5.4.1 Système mixte avec modèle à deux types d'impact ... *86*
5.4.2 Système mixte avec modèle à quatre types d'impact ... *90*
5.4.3 Système PRECI avec modèle à deux types d'impact ... *91*
5.4.4 Synthèse de l'expérience ... *93*
5.5 CONCLUSION ... 94

CHAPITRE 6 CONCLUSIONS ET PERSPECTIVES ... 96

6.1 CONCLUSIONS ... 96
6.2 PRINCIPALES CONTRIBUTIONS ... 99
6.3 PERSPECTIVES ... 99

BIBLIOGRAPHIE ... 101

ANNEXES ... I

Liste des tableaux

Tableau 1 : Liste des métriques de système de BOAP 57
Tableau 2 : Liste des métriques de classes de BOAP 58
Tableau 3 : Liste des métriques de routines de BOAP 59
Tableau 4 : Liste des métriques à implémenter........ 61
Tableau 5 : Caractéristiques des systèmes étudiés 75
Tableau 6 : Questionnaire du calcul d'impact........ 78
Tableau 7 : Statistiques descriptives des métriques OO analysées (système PRECI) 79
Tableau 8 : Statistiques descriptives des métriques OO analysées (système BOAP) 80
Tableau 9 : Occurrences des changements pour le système PRECI........ 81
Tableau 10 : Occurrences des changements pour le système BOAP 82
Tableau 11 : Occurrences des changements pour le système mixte........ 83
Tableau 12 : Variables indépendantes et variables dépendantes 84

Liste des figures

FIGURE 1 : GRAPHE DE RELATIONS ENTRE LES CLASSES 13
FIGURE 2 : PROCESSUS DU CHANGEMENT DE LOGICIEL [13] 21
FIGURE 3 : PROCESSUS D'ANALYSE D'IMPACT [48] 28
FIGURE 4 : APPROCHE DE L'ANALYSE D'IMPACT 37
FIGURE 5 : CATÉGORIES D'EFFETS DE CHANGEMENTS 47
FIGURE 6 : ARCHITECTURE GÉNÉRALE DE BOAP 54
FIGURE 7 : DIAGRAMME DE CLASSES DU PACKAGE DE MÉTRIQUES DE BOAP 56
FIGURE 8 : PROCESSUS D'ANALYSE ET D'EXTRACTION DE MÉTRIQUES 67
FIGURE 9 : ARBRE SYNTAXIQUE ABSTRAIT (AST) 68

Liste des sigles et abréviations

Acronyme	Description
AST	Abstract Syntax Tree
BOAP	Boite à Outils d'Analyse de Programmes
CRIM	Centre de Recherche Informatique de Montréal
GLIC	Génie Logiciel et Ingénierie de la Connaissance
IEEE	Institute of Electrical & Electronics Engineers
ISO	International Organization for Standardization
OO	Orienté Objet
PRECI	Planification de Ressources Électriques basée sur la Connaissance et l'Inférence.
SWEBOK	Software Engineering Body of Knowledge
WEKA	Waikato Environment for Knowledge Analysis.

Remerciements

Je dois ma reconnaissance à mes deux directeurs de recherche pour leurs appuis durant mon projet de maîtrise. Sans leur soutien, ce travail n'aurait pas été accompli.

Je tiens à remercier vivement le professeur Hakim Lounis, mon directeur de recherche à l'UQAM, pour m'avoir offert l'opportunité de passer un stage au sein de l'équipe de recherche du CRIM, pour sa disponibilité, ses directives et son appui financier.

Je tiens à exprimer mes remerciements au professeur Houari Sahraoui, mon directeur de recherche à l'UDM, pour sa disponibilité, ses conseils et ses directives si enrichissants.

Je tiens aussi à exprimer mes remerciements aux gens du CRIM, et spécialement El Hachemi Ali Kacem pour sa collaboration, ses conseils et surtout sa disponibilité.

Je remercie le professeur Yann-Gaël Guéhéneuc et le professeur Esmahi Sidi Larbi pour leurs directives, leurs aides et leurs commentaires lucratifs.

Il me tiens à cœur de remercier plus profondément mes très chers parents qui ont fait de moi ce que je suis aujourd'hui. Je les remercie pour leurs conseils, leurs encouragements, leurs sacrifices tout au long de mes études et ma vie en général, et surtout pour avoir semé en moi le goût pour la connaissance et le désir de l'acquérir.
J'adresse mes remerciements à mes frères, mes sœurs et toute ma grande famille, et qu'ils trouvent ici la reconnaissance de leurs encouragements et de leurs soutiens.

Je remercie mes amis de l'Université de Montréal, en particulier les membres du groupe GELO. Je remercie encore tous mes amis pour leurs encouragements, leurs conseils, leur soutien, et tout simplement leur présence.

Finalement, je remercie toutes les personnes qui ont contribué de prés ou de loin à la réalisation de ce travail.
Merci à vous tous.

À celui qui m'a éclairé le chemin de la science.

À la mémoire de mon père. À ma mère, mes frères et mes sœurs.

Chapitre 1 Introduction

Le standard ISO 9126, établit six caractéristiques de la qualité : la fonctionnalité, la fiabilité, l'utilisabilité, l'efficacité, la maintenabilité et la portabilité. La maintenabilité est définie par : " *the ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements"* [39]. Une fois le logiciel en opération, i.e., en phase de maintenance, il est appelé à changer et évoluer : des erreurs apparaissent, l'environnement change, et de nouveaux besoins sont formulés par les utilisateurs.

Les coûts de maintenance représentent un souci majeur aussi bien pour les gestionnaires que pour les personnes chargées de la maintenance, même s'il s'agit des systèmes conçus avec des techniques avancées telles que l'orienté objet. Dans le cas de systèmes industriels, un flot continu de changements doit être pris en compte. En effet, avec l'introduction de nouveaux changements, la taille augmente, les fonctionnalités changent, et la structure du système dérive et devient complexe, ce qui réduit la maintenabilité du système. La modification d'un système orienté objet est une tâche qui demande beaucoup de précaution. Les personnes chargées de la maintenance doivent d'une part analyser le changement afin de localiser la partie du code source à modifier, et d'autre part, déterminer les impacts de ce changement sur le reste du système. Il est donc essentiel pour un responsable de maintenance de gérer et de contrôler les tâches de maintenance. Vu qu'on ne peut gérer ce qu'on ne peut mesurer, il apparaît nécessaire de pouvoir quantifier l'impact du changement pour bien gérer la maintenance. Ainsi, il est important de maîtriser l'impact du changement afin de minimiser le coût de la maintenance.

1.1 Contexte

La maintenance a été souvent représentée comme la dernière phase du cycle de vie du logiciel. Au cours des années, plusieurs définitions ont été proposées pour décrire la maintenance de logiciel. Le standard IEEE pour la maintenance du logiciel [38] définit la maintenance par : *"software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environnement"*. D'un autre côté, le standard ISO/IEC12207 [40] décrit la maintenance comme

: *"the process of software product undergoing modification to code and associated documentation due to a problem or the need for improvement. The objectif is to modify existing software product while preserving its integrity".* Enfin, Glass et Noiseux [34] définissent la maintenance par : *"software maintenance is the act of taking a software product that has already been delivered to a customer and is in use by him, and keeping it functioning in a satisfactory way".*

Ces définitions identifient les opérations fondamentales de la maintenance (correction, adaptation et perfectionnement) qui sont effectuées afin de garder le système opérationnel, techniquement à jour et répondant aux besoins des utilisateurs. En plus de la détection et de la correction des erreurs, c'est un processus d'évolution qui permet d'assurer la continuité du logiciel à satisfaire les spécifications actuelles et les exigences nouvelles suite aux changements fonctionnels, matériels et logiciels. Ce processus est défini par Arthur [8] comme un changement continu à partir d'un état faible, simple ou mauvais vers un état plus haut et meilleur.

Par ailleurs, la maintenance du logiciel représente la phase la plus importante et la plus coûteuse, 60% à 80% du coût total du cycle de vie du logiciel est dépensé durant la phase de maintenance [58]. Ceci est dû entre autres (i) au personnel inexpérimenté, non familier avec le système et parfois peu motivé, (ii) aux programmes non organisés, n'obéissant pas aux standards, (iii) aux modifications qui génèrent des fautes, (iv) à la dégradation de la structure du système suite aux modifications non planifiées apportées au système par des gens différents sur des plates-formes différentes sans documentation de leurs tâches, et (v) à une documentation obsolète, inadéquate, voire inexistante.

De nos jours, la maintenance reçoit plus d'intérêt de la part des chercheurs dans le but de maîtriser ses coûts élevés et d'augmenter la maintenablité des systèmes. Le problème de l'an (Y2K)2000, l'expansion du paradigme "Open Source" et le besoin des organisations d'amortir la majeure partie de leur investissement de développement en maintenant le logiciel opérationnel le plus longtemps possible, ont apporté plus d'attention à cette phase [66]. Ainsi, plusieurs techniques, méthodes et outils destinés à cette phase ont vu le jour. On cite comme exemple : la réingénierie, la rétro ingénierie, l'analyse d'impact, les tests de régression, la gestion de la configuration de logiciel, la maintenance via le Web, etc.

Dans ce travail, nous étudions l'analyse d'impact du changement dans le but de supporter la maintenance des systèmes orientés objets.

1.1.1 Maintenance des systèmes

La compréhension d'un programme représente l'une des tâches les plus importantes du processus de maintenance. Les personnes chargées de la maintenance épuisent 40 à 60% de leurs temps à lire le code et à essayer de comprendre sa logique et son architecture. C'est une activité laborieuse qui augmente le coût de la maintenance [61]. En effet, l'équipe qui a développé le système n'est pas toujours celle qui assure sa maintenance, les documents de spécifications sont parfois incomplets et le code source constitue souvent la seule source d'informations fiable. La plupart des problèmes associés à la maintenance du logiciel sont causés par les méthodes utilisées pour concevoir et développer le système. De plus, la maintenabilité est un facteur de qualité qui n'est pas incorporé dans le processus de développement. Pour faciliter la maintenance et augmenter la maintenabilité des systèmes logiciels, il faut, entre autres, définir des standards de codage, de documentation et d'outils de tests dans la phase de développement du logiciel [66], documenter l'évolution du logiciel et utiliser les techniques destinées à la maintenance.

1.1.2 Techniques de maintenance

Dans le domaine du logiciel, une plus grande partie de l'effort (avant la décennie quatre vingt) était accordée à la phase de développement. L'objectif était de livrer un produit qui satisfait les besoins dans les délais et dans les limites du budget sans se préoccuper de la qualité du logiciel. Ce n'est qu'en 1983 que la communauté des chercheurs a tenue la première conférence sur la maintenance (International Conference on Software Maintenance : ICSM1983). Depuis, de nombreuses techniques sont utilisées pour maîtriser les coûts élevés de la maintenance. Ci-dessous, nous présentons brièvement quelques-unes de ces techniques.

Rétro ingénierie : généralement pour concevoir un logiciel, on passe de la conception au code. La rétro ingénierie prend le chemin inverse. Chikofsky et Cross [27] ont défini la rétro ingénierie par: *"reverse engineering is the process of analyzing a subject system to identify the system's component and their interrelationships and create representations of the system in another form or at a higher level of abstraction"*. C'est le processus d'analyse d'un logiciel pour identifier ses composants et leurs relations dans le but de générer une représentation du système à un plus haut niveau d'abstraction. La redocumentation et la récupération de la conception (Design recovery) représentent deux types importants de rétro ingénierie.

Réingénierie : selon Chikofsky et Cross, la réingénierie consiste en : *"the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation*

of the new form "[27]. C'est un processus composé de l'analyse du système et puis sa reconstitution. Ces deux étapes sont effectuées par les techniques de rétro ingénierie et de *"Forward Engineering"*. Cette dernière consiste à transiter d'une représentation de haut niveau obtenue par l'étape précédente pour produire un nouveau système. Certains auteurs ajoutent une troisième étape : la restructuration. Elle transforme la structure interne d'un logiciel sans changer de niveau d'abstraction et sans modifier ou ajouter de nouvelles fonctions. En outre, elle permet une meilleure compréhension du logiciel et facilite sa maintenance [1].

Analyse d'impact : pratiquement, lorsqu'une proposition de changement parvient à l'équipe de maintenance, elle est transformée en terme logiciel afin de décider si le problème doit être retenu ou rejeté. Une fois accepté, il faut localiser l'origine du changement. L'analyse d'impact du changement s'impose à cette étape afin de (i) déterminer tous les composants du système qui doivent changer en conséquence du changement initial, (ii) estimer les ressources nécessaires pour implanter le changement. Le résultat de cette analyse permet aux gestionnaires de prendre une décision sur la meilleure solution à mettre en œuvre ou annuler la requête de changement.

1.1.3 Modèles de maintenance

Depuis des années, plusieurs modèles de maintenance de logiciel ont été proposés, pour souligner des aspects particuliers de la maintenance de logiciel. Mc Clure propose un modèle composé de trois phases dont la compréhension, la modification et la revalidation du programme [56]. Il montre l'utilité de chaque phase et fournit des explications sur comment effectuer les tâches de maintenance. Yau et Collofello [72] soulignent quatre phases dans la maintenance du logiciel. Leur modèle s'intéresse à la stabilité du logiciel à travers l'analyse de la propagation du changement du logiciel. Arthur fournit un modèle plus compréhensible permettant de traiter les requêtes de changement jusqu'à ce qu'elles soient implémentées, testées et délivrées aux utilisateurs [8]. Il introduit la notion d'analyse d'impact comme étape importante avant d'implémenter le changement. Enfin, Pfleeger propose un modèle qui se concentre sur l'analyse d'impact et les métriques pour évaluer et contrôler le changement [59]. L'analyse d'impact du changement avec des métriques permet aux gestionnaires de confirmer si le changement répond aux exigences, est conforme à la conception existante, ne dégrade pas la maintenabilité du système existant, et est implémenté de la meilleure manière.

Après avoir présenté les divers modèles de maintenance, nous remarquons la présence de l'analyse d'impact du changement comme étape importante dans le processus de maintenance. Ainsi, pour effectuer un changement dans un système en maintenance, il faut d'abord acquérir

une familiarité avec ce système. Cette familiarité se concrétise par la compréhension du code source, sa structure, ses fonctions et son traitement. Étant acquises, ces informations servent de base pour effectuer l'analyse d'impact du changement au cours de laquelle, les composants affectés sont décelés, les ressources nécessaires sont estimées et les solutions possibles pour accomplir le changement dans les bons délais sont fournies. Une fois les modifications implantées et le système revalidé, la nouvelle version du système est livrée. En pratique, l'équipe de maintenance travaille en collaboration avec les utilisateurs et les opérateurs du système afin d'unifier l'objectif du changement et ainsi d'éviter les ambiguïtés.

1.2 Problématique et Motivations

L'introduction de la technologie orientée objet dans le domaine de l'ingénierie du logiciel permet de produire un code de qualité grâce à la notion de classe qui représente une unité de modularité naturelle [64]. De plus, elle a pour bénéfice la production de systèmes avec une meilleure maintenabilité et réutilisation [20]. Cependant, malgré ces avantages, elle ne peut assurer la qualité du logiciel ni éviter les erreurs introduites par les programmeurs, que cela soit durant le développement ou la maintenance ou lors de modifications.

Si un logiciel est facilement changeable, d'une part on peut le modifier rapidement, fréquemment, avec moins de ressources et à un coût relativement bas. Cependant, tant qu'on le modifie, il peut augmenter en taille et complexité, et la dégradation de sa structure le rend ainsi fragile et difficile à comprendre, à changer et à maintenir. En effet, les changements sont effectués la plupart du temps sans planification préalable, c'est seulement quand le souci de faire un nouveau changement devient sérieux qu'on commence à évaluer l'impact et le coût du changement avant de l'implémenter.

Briand *et al.* précisent [19] que la production de meilleures spécifications et conceptions réduit le besoin de revue, de modification et de réécriture non seulement du code, mais aussi des spécifications et des documents de conceptions. En conséquence, l'organisation possédant le logiciel peut économiser du temps, réduire les coûts de production et augmenter la qualité du produit final. La disponibilité précoce de valeurs de métriques est un facteur clé pour une bonne gestion de développement du logiciel, puisqu'elle permet :

- une détection précoce des problèmes dans les composants (artefacts) produits dans les phases initiales du cycle de vie (documents de spécifications et de conceptions), et donc

une réduction du coût du changement (l'identification et la correction tardives des problèmes sont plus coûteuses);

- une meilleure surveillance de la qualité du logiciel dès les phases initiales du cycle de vie du logiciel ;
- une comparaison quantitative des techniques et amélioration empirique du processus auquel elles sont appliquées ;
- Une planification plus précise de l'allocation des ressources, basée sur la prédiction de la maintenabilité du système et des parties qui le constituent.

Rombach dans son étude basée sur les mesures, décrit que les mesures de conception peuvent être utilisées dans le but de prédire la maintenabilité [63]. Il distingue deux types de conception : architecturale et algorithmique (high level/low level). La conception architecturale permet d'identifier les composants du système et leurs interconnections, la conception algorithmique identifie les structures de données et le flot de contrôle à l'intérieur des composants architecturaux. Selon lui, le niveau architectural a plus d'influence sur la maintenabilité que le niveau algorithmique.

La maintenabilité est déterminée par la manière avec laquelle un système supporte les tâches de maintenance dont l'analyse d'impact et les tests de régression [20]. Avec l'évolution des systèmes logiciels, un flot important de changement doit être pris en considération ainsi que leur propagation sur le reste du système. Réussir à modifier le logiciel de façon disciplinée toute en maintenant son fonctionnement et son intégrité avec un coût raisonnable nécessite une analyse de l'impact du changement avant modification. En effet, l'équipe de maintenance doit être en mesure de fournir des réponses aux questions suivantes :

1. De quel type de changement s'agit-il ?
2. Quelle est l'étendue du changement ?
3. Quelles sont les ressources nécessaires pour faire le changement : temps, personnels, budget et plannings ?
4. Quel est le degré de difficulté à implanter le changement ?
5. Quel est le risque encouru en accomplissant le changement ?

De telles informations aident le responsable de la maintenance dans l'évaluation du coût de la proposition de modification et représentent une source cruciale pour les gestionnaires afin de planifier le changement, et déterminer les ressources nécessaires pour l'implémenter.

1.3 Objectifs

L'objectif de ce mémoire est d'améliorer la maintenabilité des systèmes logiciels orientés objets via l'estimation de l'impact du changement et de ses conséquences sur le reste du système : l'effet de propagation (ripple effect). Cet objectif se compose des volets suivants :

1. Mesurer l'impact du changement, au niveau des classes, des programmes orientés objets écrits en Java et étudier l'influence de la structure sur l'impact du changement. Pour cela, nous avons proposé deux hypothèses qui reflètent l'influence des propriétés de conception orientée objet sur l'impact du changement, en particulier le couplage et l'héritage.
2. Choisir des changements pour lesquels nous voulons estimer les impacts. Pour cela, nous avons réalisé une expérience en vue de classifier les changements selon leur chance de se présenter dans les systèmes à objets sur deux systèmes avec la collaboration du CRIM.
3. Définir notre modèle d'impact pour mesurer l'impact du changement. Ce modèle d'impact est composé de quatre niveaux qui varient selon le degré d'influence du changement sur le reste du système.
4. Mener une expérience pour estimer l'impact du changement. Pour cela, nous avons établi un questionnaire de calcul d'impact. Ce questionnaire est composé des types de changements que nous avons choisi dans l'étape 2 et utilise le modèle d'impact de l'étape 3 sur deux systèmes fournis par le CRIM.
5. Choisir et extraire des métriques capturant les caractéristiques importantes comme le couplage et l'héritage. Nous avons exploré plusieurs outils d'extraction de métriques, nous avons choisi BOAP pour ces fins. Cependant, certaines métriques d'implémentation ne peuvent être fournies par BOAP, nous avons donc procédé au développement d'un nouvel outil pour Eclipse. L'objectif visé en collectant ces métriques est de faire ressortir les mesures de conception qui sont de bons indicateurs d'impact de changement.
6. Utiliser les algorithmes d'apprentissage pour construire nos modèles prédictifs afin d'étudier les relations suggérées par les hypothèses établies dans l'étape 1 et faire ressortir les métriques qui sont de bons indicateurs d'impact de changement. Ces modèles sont construits à base des métriques calculées dans l'étape 5 et des résultats du questionnaire du calcul d'impact de l'étape 4. Enfin, analyser et interpréter les résultats et dresser les conclusions de ce mémoire.

1.4 Organisation du mémoire

Ce mémoire est organisée comme suit : le chapitre 2 présente l'état de l'art sur l'analyse d'impact du changement et introduit quelques concepts de base de l'approche orientée objet; il sera suivi d'une étude de quelques métriques orientées objets proposées dans la littérature. L'approche d'analyse d'impact est détaillée dans le chapitre 3 incluant la proposition des hypothèses à valider, la classification des changements que peut connaître un code source Java, la définition du modèle d'impact du changement et la méthode utilisée pour l'analyse de données. Le chapitre 4 introduit les deux plates-formes qui ont servies pour l'extraction et le calcul des métriques : BOAP (Boite à Outils d'Analyse de Programmes) et l'outil que nous avons développé pour la plate-forme Eclipse. Le chapitre 5, présente l'expérience réalisée, y compris le choix de métriques, la liste des changements, les systèmes sous tests et l'analyse des résultats. Le chapitre 6 clos ce mémoire et dresse des conclusions.

Chapitre 2 État de l'art

Dans le chapitre précédent (section 1.1.2), nous avons cité un ensemble de techniques de maintenance de logiciel qui contribuent à la réduction du coût de cette phase, augmentent la productivité des programmeurs, facilitent la tâche du changement et dotent les gestionnaires d'outil d'aide à la décision. L'analyse d'impact du changement est le sujet de notre projet de mémoire. Ce chapitre est réparti en trois grandes parties. Dans un premier temps, on présente des notions de base en relation avec l'analyse d'impact et le paradigme orienté objet. Ensuite, on introduit l'analyse d'impact du changement, en particulier pour les systèmes orientés objets. Enfin, en dernier lieu, on traite des métriques orientées objets et de leur utilité pour déterminer l'impact du changement. Un ensemble de travaux de recherche dans ce domaine sont aussi présentés et discutés.

2.1 Notions de Base

2.1.1 Concepts de l'approche orientée objet

L'application de la technologie orientée objet produit des systèmes logiciels bien structurés, avec des architectures compréhensibles qui sont faciles à tester, à maintenir et à étendre [64]. Les concepts véhiculés par l'approche orientée objet sont importants à considérer dans une problématique comme la nôtre. Un système orienté objet est composé d'*objets* et de *classes*. Un objet est caractérisé par un état et un comportement. L'*état* englobe toutes les propriétés caractérisant l'objet ainsi que la valeur de chacune des propriétés. Le *comportement* représente comment un objet réagit et interagit en fonction de son changement d'état et d'échange de messages [15]. La *classe* est une spécification de l'objet, c'est le modèle ou le prototype de base à partir duquel les objets sont crées. Les classes sont définies par les services publics offerts aux utilisateurs par l'intermédiaire des méthodes et attributs publics. Les *méthodes* implémentent les opérations qu'on peut effectuer dans une classe (la logique de programmation). Les *attributs* sont définis par leurs noms et leurs types simples ou complexes (int i ou Class_A i). Le

mécanisme de *protection* détermine la visibilité et par conséquent l'accessibilité aux entités du logiciel à partir d'autres classes. Elle peut être publique, protégée ou privée.

L'approche orientée objet est basée sur plusieurs concepts dont l'encapsulation, l'héritage et le polymorphisme. L*'encapsulation* est une manière de séparer l'implémentation d'une classe de sa spécification. C'est le fait de cacher ou masquer tous les détails d'un objet qui ne contribuent pas essentiellement aux caractéristiques de l'objet [15]. L'utilisateur connaît ce que l'objet fait (son comportement) mais pas comment il est fait (son implémentation). Puisque toutes les propriétés de l'objet sont cachées, son interface constitue le seul moyen de communication avec lui via l'invocation de ses propriétés publiques. *L'héritage* correspond à la création de nouvelles classes (classes dérivées) en se basant sur les classes existantes (classes de bases) soit par réutilisation ou par extension des fonctionnalités de ces dernières. Une classe dérivée hérite de la structure et du comportement de la classe de base et peut servir de classe de base pour une dérivation ultérieure donnant naissance à une hiérarchie de classes appelée *arbre d'héritage*. Le *polymorphisme* signifie formes multiples. Autrement dit, une méthode peut faire différentes choses dépendant de la classe qui l'implémente. La liaison dynamique (dy*namic binding)* est le processus de lier un appel de méthode à une méthode particulière ; ceci est effectué dynamiquement (en Java mais pas toujours en C++) au moment de l'exécution.

2.1.2 Notion de dépendance

Une dépendance dans un système logiciel est une relation directe entre les entités dans le système X → Y tel que le programmeur qui modifie X doit être concerné par les effets de bord (side effects) possibles dans Y [70].

Wilde et Huitt [70] ont classifié les dépendances en : (i) dépendances de données entre deux variables, (ii) dépendances d'appels entre deux modules, (iii) dépendances fonctionnelles entre un module et les variables qu'il calcule, et (iv) dépendances de définition entre une variable et son type. Loyall [53] répartit les dépendances en deux types : (i) dépendance inter-procédures, à titre d'exemple, une procédure Ma est dépendante d'une autre Mb, si et seulement si au moins une instruction (statement) de Ma dépend d'une instruction de Mb, et (ii) dépendances intra-procédure en tenant compte des dépendances entre les différentes instructions à l'intérieur d'une procédure. Lee, dans sa thèse, se base sur la classification des dépendances en orienté objet de Wilde et Huitt et présente cinq catégories de dépendances : (i) dépendances classe à classe, (ii)

dépendances classe à méthode, (iii) dépendances classe à variable, (iv) dépendances méthode à variable, et (v) dépendances méthode à méthode [48].

Cette classification de dépendances se base sur les relations existantes parmi les éléments d'un système orienté objet telles que l'héritage, l'agrégation et l'utilisation. Ces relations impliquent certainement qu'une classe dépend d'une autre classe. Par exemple, la relation d'héritage implique que la classe dérivée réutilise les attributs et les méthodes de la classe de base. Par conséquent, elle dépend de la classe de base. De façon similaire, une classe peut contenir des instances des autres classes, et une méthode peut utiliser des variables d'instances ou de classes.

Une classe peut être directement liée avec une autre via un des liens cités précédemment ou indirectement par une succession de liens entre les classes du programme. De plus, la représentation graphique des classes constituant un système avec ces liens permet une bonne compréhension de son architecture. Dans les sections qui suivent, on présente ces liens et ces graphes.

2.1.2.1 Relations directes entre classes

Une relation directe entre deux classes se manifeste par l'existence d'au moins une des relations de dépendances suivantes : l'agrégation, l'utilisation ou l'héritage.

Agrégation : Représente la relation de tout/partie. Selon Li et Offut [50], c'est une relation de contenance. À titre d'exemple, on dit qu'une voiture a des portes ou un avion possède des ailes.

On dit que la classe Class_A est une agrégation de la classe Class_B si :

- Un attribut ou plus de Class_A est une instance de Class_B.
- Un tableau d'instances de Class_B est un attribut de Class_A.
- Un attribut de Class_A est une référence vers une instance de Class_B.
- Un attribut de Class_A est un tableau de références vers des instances de Class_B.

Les deux premiers types correspondent à l'agrégation automatique et les deux derniers correspondent à l'agrégation dynamique [47].

Utilisation : On dit que Class_A utilise Class_B, si Class_A (ou ses instances) envoie des messages à Class_B (ou ses instances) [73]. Class_A a besoin de fonctionnalités offertes par Class_B, appelée client de Class_B, et Class_B offre ses services à Class_A, appelée serveur de Class_A. Ci-dessous des exemples d'utilisation :

- Class_A invoque une méthode de Class_B.
- Le type de retour d'une méthode de Class_A est une instance de Class_B.
- Une instance de Class_B est passée en paramètre dans une méthode de Class_A.

- Une variable de type Class_B est modifiée ou utilisée par une méthode de Class_A.
- Un attribut de Class_B est modifié ou utilisé dans une méthode de Class_A.

Héritage : Le concept d'héritage correspond à la création de nouvelles classes d'objets en se basant sur les classes existantes soit par réutilisation ou par extension des fonctionnalités de ces dernières. Lee et Barby présentent trois schémas d'héritage dépendant des caractéristiques de la classe dérivée par rapport à la classe parente [9, 48].

- "Strict inheritance" est le schéma le plus simple de l'héritage. La classe drivée garde le comportement exact hérité de sa classe parente. Les propriétés héritées ne peuvent être modifiées (overriden), et la classe dérivée peut être raffinée seulement par l'ajout de nouvelles propriétés.
- "Subtyping" est le plus courant type d'héritage. En plus des propriétés de l'héritage strict, il permet la redéfinition des propriétés héritées quand les opérations de la classe parente ne sont pas appropriées pour la sous-classe (donner une nouvelle implémentation à une opération héritée pour répondre aux besoins propres de la sous-classe).
- Lorsque la classe dérivée n'est pas une spécialisation de la classe de base et représente une abstraction complètement nouvelle (qui base une partie de son comportement sur une partie d'une autre classe), ce type d'héritage est appelé "Subclassing".

2.1.2.2 Relations indirectes

Une classe peut dépendre indirectement d'une autre classe via une séquence successive de dépendances intermédiaires. Une classe Class_A a une relation indirecte avec une classe Class_B notée par Class_A **R+** Class_B, s'il existe une chaîne :

Class_B1, Class_B2, Class_B3.................................. Class_Bn, avec n >=2,

tel que **Class_A R** Class_B1, Class_B1 **R** Class_B2................. Class_Bn **R Class_B**.

2.1.2.3 Graphe Orienté

Un graphe orienté (directed graph) permet de représenter les dépendances entre les classes d'un système à objets et de donner une vue globale de l'architecture du programme. Un graphe orienté, noté par G, est l'ensemble G= (Ng, Eg), avec Ng un ensemble fini de nœuds, et Eg un ensemble fini d'arcs $Eg \subseteq (Ng \times Ng)$. Pour chaque arc $(u, v) \in Eg$, u est la source et v est la destination. Un chemin dans un graphe G est une séquence finie non nulle de nœuds Q = n1, n2, ...nk, avec chaque $ni \in Ng$ pour i = 1...k et chaque couple $(nj, nj+1) \in Eg$ pour j = 1...k-1. Q est appelé chemin de n1 à nk, et k est appelé la longueur de Q [53].

2.1.2.4 Graphe de relation

Un graphe de relation permet de capturer toutes les relations de dépendances entre les différentes classes d'un système orienté objet afin de mieux comprendre les interactions et la structure de ce système. Ce graphe est utilisé par différents auteurs dans leurs travaux de recherche avec des appellations différentes. Pour Kung *et al.*, il s'agit "d'Object Relation Diagram" [47] et pour Lee, c'est "Object Oriented System Dependency Graph" [48].

En général, un graphe de relation est un graphe orienté G= (N, D) où N est l'ensemble des nœuds représentant les classes du programme N = {Class_C1, Class_C2, Class_C3, ...Class_Cn}, et D l'ensemble des relations de dépendances entre ces classes D = {U,G,H} : avec l'héritage (H), l'utilisation (U) et l'agrégation (G). La figure ci-dessous est un exemple de ce flux de relations de dépendances dans un système orienté objet.

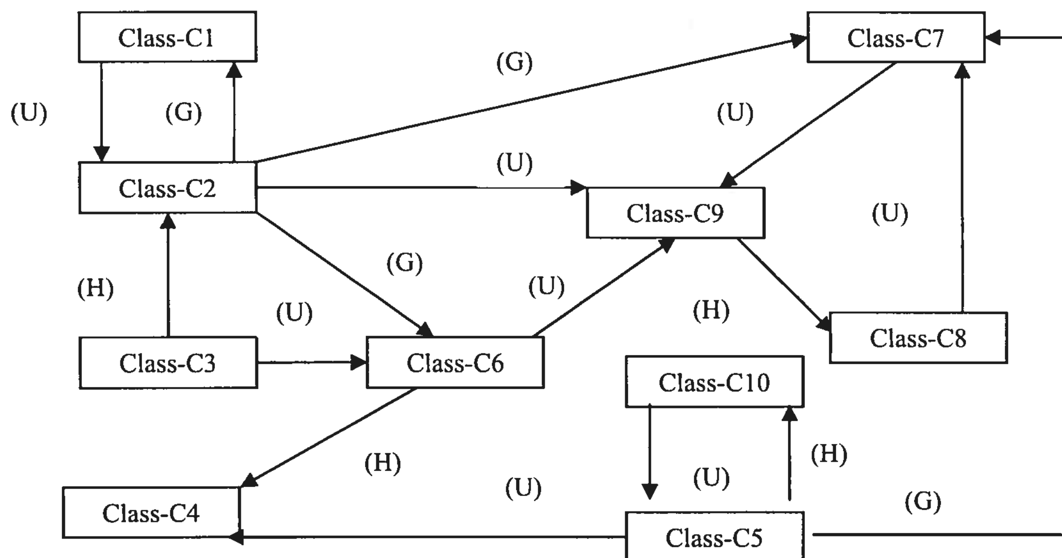

Figure 1 : Graphe de relations entre les classes

2.1.3 Notion d'impact

Un impact représente l'effet ou la conséquence d'une chose sur une autre. Arnold et Bohner considèrent l'impact comme : *"a part determined to be affected, and therefore worthy of inspection"*[14]. Dans les systèmes orientés objets, l'unité de base est la classe. On dit qu'une classe Class_A a un impact sur une autre Class_B, si un changement dans Class_A altère directement ou indirectement le comportement de Class_B.

La notion d'impact est étroitement liée à la notion de dépendances (section 2.1.2). Ainsi, si Class_B dépend de Class_A directement ou indirectement, alors un changement dans Class_A

affecte Class_B. Si une classe est dépendante de beaucoup d'autres classes, alors les changements dans les autres classes peuvent affecter la classe dépendante. Ce qui rend sa réutilisation difficile, et sa séparation de son environnement ne peut se faire sans altérations supplémentaires.

Lee divise les types d'impact en deux dimensions : impact syntaxique et sémantique, et impact statique et dynamique [48]. L'impact syntaxique est calculé purement par des informations extraites du code source telles que le flux de données (data flow), le flux de contrôle (control flow) et la hiérarchie des appels. L'impact sémantique se base sur les connaissances sémantiques pour détecter les effets de bord possibles. Les informations sémantiques sont plus difficiles à dériver et à vérifier. L'impact statique est calculé selon l'information statique obtenue au moment de la compilation. L'impact dynamique est calculé sur l'exécution du programme. Robert considère deux aspects d'impact : quantitatif et qualitatif. Le premier consiste à compter le nombre d'éléments touchés et le deuxième détermine la manière avec laquelle ces éléments sont touchés [62].

Comme pour les dépendances, l'impact entre les classes d'un système orienté objet peut être direct ou indirect. L'impact direct est causé par la relation de dépendance directe entre la classe changée et la classe affectée. L'impact indirect est causé par l'existence d'une chaîne de relations entre une série de classes.

2.1.4 Propagation d'impact du changement

Nous avons conclu la section précédente par deux types d'impact : direct et indirect. L'impact direct est détecté automatiquement par les relations directes de dépendances (section 2.1.2.1) qui lient la classe changée avec les autres classes du système. De ce fait, si une classe Class_A est liée par une des relations de dépendances avec une autre classe Class_B, alors tout changement dans Class_A aura un impact sur Class_B. Pour déterminer l'impact indirect connu sous le nom d'effet de propagation, la fermeture transitive et le firewall sont utilisés. Ces notions de propagation d'impact du changement sont exposées dans les sections suivantes.

2.1.4.1 Fermeture transitive

La relation R+ (section 2.1.2.2) est définie par :

R+ = {(Class_A, Class_B) tel que ∃ Class_B1, Class_B2… Class_Bn, avec Class_ARClass_B1, Class_B1RClass_B2, …., Class_BnR Class_B}. Elle inclut la chaîne des

classes qui sont reliées directement ou indirectement par la relation R. Elle est appelée fermeture transitive (transitive closure). Dans un graphe orienté, déterminer pour un nœud initial (Ni) tous les nœuds cibles (Nc) du graphe pour lesquels il existe un chemin de Ni vers Nc en suivant les arcs, consiste à calculer la fermeture transitive résolu par l'algorithme de Warshall [68].

Du point de vue impact, D est l'ensemble des classes reliées directement ou indirectement à la classe Class_C représenté par : {D tel que Class_C **R+** D}. Si Class_C change, alors l'ensemble des classes sur lesquelles elle peut avoir un impact est D. Cet ensemble est connu sous le nom de firewall de Class_C. Ce dernier permet de recenser les classes affectées par une modification dans la classe initiale ou détecter les classes à retester lors des tests de régression.

2.1.4.2 Effet de propagation "Ripple effect"

Bohner définit l'effet de propagation par : " *effect caused by making a small change to a system which impacts many other parts of a system*" [14]. L'effet de propagation d'un changement apporté au code source d'un système logiciel est défini par Turver comme *"the consequential effects on other parts of the system resulting from that change. These effects can be classified into a number of categories such as logical effects, performance effects, or understanding effects"* [67].

L'effet de propagation peut également se présenter pendant les phases initiales du cycle de vie d'un logiciel orienté objet (conception et spécification). Par exemple, un changement dans les spécifications du système affecte les autres phases du cycle de vie du logiciel dont le développement, le test et la maintenance. Cette propagation se manifeste dans les plannings, les délais de livraisons, le budget, les ressources humaines, etc.

Par ailleurs, un changement dans la logique de programmation d'une classe peut affecter les fonctions ou le comportement des autres classes auxquelles elle est liée. Michelle Lee propose deux types d'effet de propagation [48] :

- Effet de propagation direct (ripple effet direct) : se présente lorsque le changement d'une variable affecte directement la définition d'une autre variable.
- Effet de propagation indirect (ripple effect indirect) : se produit quand la variable affectée à son tour affecte d'autres variables.

La stabilité du logiciel réside dans sa résistance aux conséquences des changements. Selon Turver, l'analyse de la stabilité diffère de l'analyse d'impact parce qu'elle considère la somme

des effets de propagations possibles plutôt qu'un effet de propagation particulier causé par un changement [67].

2.1.4.3 Firewall

Le concept de firewall dans le contexte des systèmes orientés objets a été proposé par Kung *et al.* [46]. Le firewall de la classe Class_C noté par CFW (Class_C) correspond à l'ensemble des classes qui peuvent être affectées par un changement dans Class_C. Autrement dit, toutes les classes sur lesquelles Class_C peut avoir un impact. Le firewall représente la portée maximale d'un changement. Il contient toutes les classes possiblement mais pas nécessairement affectées par le changement. En vue de s'assurer de la validité des modifications apportées au système, il faut retester toutes les classes se trouvant dans le firewall.

La construction du firewall se base sur la notion de la fermeture transitive (section 2.1.4.1). Le firewall de la classe Class_C est défini par : CFW(Class_C) = {Class_Cj | (Class_C, Class_Cj) $\in$ **R+**}. Cette notion peut être étendue pour un ensemble de classes modifiées. Soit l'ensemble des classes changées noté par S={Class_C1, Class_C2......Class_Ck}. Le firewall de S est l'union des firewalls de toutes les classes qui se trouvent dans l'ensemble S noté par :
CFW(S) = $\cup$ CFW (Class_Ci) $\forall$ Class_Ci $\in$ S pour i =1......k.

2.1.5 Tests de régression

L'approche orientée objet doit permettre de produire une conception propre, bien comprise et facile à maintenir. Cependant, bien qu'une conception orientée objet puisse aboutir à une meilleure architecture de système et qu'un langage de programmation orienté objet impose un style de codage discipliné, on ne peut éviter les erreurs des programmeurs et le manque de compréhension des spécifications [9]. Tester un système lors de la phase de développement et avant livraison permet de s'assurer de son bon fonctionnement et qu'il répond aux besoins des utilisateurs. Mener ces tests lors de la maintenance permet de déterminer les erreurs introduites par les programmeurs lors de la modification du système (maintenance corrective, adaptive, perfective), et d'avoir confiance du fait que le système continue à satisfaire ses spécifications.
Les tests de régression s'effectuent au moment de la modification et portent sur la partie du code modifiée (test unitaire) et quand toutes les modifications sont complétées (tests d'intégration). Par ailleurs, ils se distinguent des tests de développement par la disponibilité au moment des tests de régression des suites de tests existantes. Utiliser ces suites de tests pour retester le programme modifié permet de réduire l'effort nécessaire pour faire ce test. Malheureusement, la

réexécution des suites de tests peut nécessiter beaucoup de temps. Le choix d'un sous-ensemble approprié d'une suite de tests existante est nommé "selective retest problem" et la méthode pour résoudre ce problème est nommé "selective retest method" [36].

Pendant la maintenance, en plus des suites de tests existantes, de nouveaux tests sont introduits (si nécessaire) pour valider les parties qui sont nouvelles ou modifiées.

2.1.5.1 Processus de tests de régression

Kung propose un processus de tests de régression qui prend en considération l'effet de propagation [46]. Les étapes de ce processus sont comme suit :

1. Identification des classes changées : la personne chargée du test compare le programme original et le programme modifié pour identifier les classes qui sont changées. Ceci peut être facilement accompli en utilisant un comparateur de code;
2. Identification des classes affectées : le testeur identifie les classes affectées. Ceci est accompli en calculant les firewalls (section 2.1.4.3) des classes qui sont changées;
3. Génération de l'ordre de test des classes : le testeur génère un ordre de test des classes pour faciliter la préparation de cas de tests et la conduite effective des tests de régression;
4. Sélection de cas de tests : le testeur sélectionne les cas de tests pour retester les classes affectées. Ceci est fait selon l'ordre de test et par consultation du plan du test s'il en existe un. L'ordre de test spécifie quelles classes doivent être testées avant quelles autres classes (ordre de priorité). Il facilite aussi la réutilisation des cas de tests;
5. Génération et modification des cas de tests : le testeur modifie les cas de tests sélectionnés et génère de nouveaux cas selon le changement effectué;
6. Exécution des tests de régression : le testeur prépare dans cette étape les données de tests selon les cas de tests et exécute le programme modifié en utilisant les données de tests. Le programme modifié est testé à différents niveaux : test unitaire de la classe, test d'intégration de la classe et test du système.

2.1.6 Conclusion

Dans cette section, nous avons présenté des concepts de base utilisés lors de l'analyse d'impact du changement. Puisqu'on s'intéresse aux systèmes développés avec le paradigme orienté objet, des notions de base ont été citées dont la classe, l'encapsulation, etc. Représenter un système par un graphe permet d'avoir une idée sur sa structure et les interactions entre ses classes via les liens de dépendances comme l'héritage, l'utilisation et l'agrégation.

Un changement peut se propager et atteindre les autres classes du système ; il s'agit de l'effet de propagation. Modifier une classe peut avoir des effets sur les classes qui lui sont reliées (i) directement à travers les relations directes ou (ii) indirectement via la succession de relations directes entre les paires de classes. Pour comprendre la propagation d'une modification, un ensemble d'outils a été présenté permettant de cerner aussi bien les classes qui sont affectées et qui nécessitent un changement, que les classes dont le comportement peut être altéré ou qui feront l'objet de tests de régression.

L'analyse d'impact du changement, sujet de ce mémoire, permet d'estimer les conséquences du changement sur le reste du système et de fournir aux gestionnaires les informations utiles pour planifier et implémenter le changement. Dans la section suivante, nous allons introduire l'analyse d'impact du changement, le processus du changement, l'analyse d'impact, les avantages et les difficultés de l'analyse d'impact, les travaux de recherche effectués dans le domaine, particulièrement ceux relatifs aux systèmes à objets, et enfin les démarches d'analyse d'impact.

2.2 Analyse d'impact du changement

Lors de la maintenance, prédire où et comment un changement affecte le reste des classes du système est une tâche difficile. Cette difficulté émane de la manière avec laquelle les modifications sont traitées. Effectuer des changements sans compréhension de leurs effets peut amener à une mauvaise estimation de l'effort, à prolonger les délais de livraison, à dégrader la conception d'un logiciel, à entamer la fiabilité du produit, voire le retrait prématuré du logiciel [48].

En effet, un changement peut apparaître simple, facile, nécessitant moins de temps, de budget et de personnel. Cependant, lors de son implémentation la personne chargée de la maintenance s'aperçoit que cette tâche est complexe, difficile, voire impossible dans certains cas. Ainsi, l'équipe de maintenance a besoin de moyens et de mécanismes permettant d'analyser le changement, de déterminer son impact sur le reste du système et d'étudier sa faisabilité. L'analyse d'impact du changement est née d'une telle préoccupation. Par exemple, pour résoudre le problème de changement de date de Y2K (passage à l'an 2000), une analyse d'impact du changement s'est avérée efficace et bénéfique en temps et effort nécessaires pour effectuer les changements. Sans cette analyse, l'équipe de maintenance doit parcourir le code

source manuellement, détecter les variables à changer, implanter les modifications, tester la conformité des changements et vérifier la cohérence du système afin de faire ressortir les conséquences de ces changements.

L'analyse d'impact du changement est effectuée principalement lors de la maintenance des programmes, en vue d'évaluer les effets du changement sur le reste du système et de réduire le risque de s'embarquer dans des dépenses coûteuses. Cette évaluation englobe l'estimation des ressources humaines, financières, temps, effort et plannings nécessaires pour accomplir le changement.

2.2.1 Processus du changement

"Software is supposed to change – otherwise software functions would be implemented in hardware" [13].

On entend par changement, un passage d'un état faible vers un autre meilleur. En génie logiciel, particulièrement en maintenance, un changement représente l'ensemble des modifications à appliquer par rapport à une situation antérieure qui vise à améliorer et à faire évoluer un système logiciel. Pour un système orienté objet, le changement peut s'appliquer aux niveaux classe, méthode ou attribut. Par ailleurs, un changement quelque soit sa nature, simple ou complexe, large ou petit, important ou trivial, de correction ou d'adaptation, de prévention ou de perfectionnement, influence l'effort d'implémentation du changement et le coût de la maintenance. La compréhension des facteurs influençant le coût de la maintenance dépend en grande partie de la compréhension des différentes catégories de maintenance [66] et de la manière avec laquelle les demandes de changements sont traitées [67]. Dans les sections suivantes, nous présentons les catégories de maintenance et le processus du changement.

2.2.1.1 Catégories de maintenance

Généralement, trois types de maintenance se distinguent, dépendant de la nature de la modification : corrective, perfective et adaptative [8, 56, 60].

Maintenance corrective : déclenchée par la détection des erreurs ou de défauts dans le fonctionnement du système. Ce type de maintenance permet de corriger les erreurs omises ou non détectées lors de la phase de test. Elle vise à éliminer les défaillances d'implémentation, les échecs du traitement et d'exécution, et assurer le bon fonctionnement du système. À titre d'exemple, la correction des programmes qui ne fonctionnent pas et ceux qui produisent de faux résultats.

Maintenance adaptative : provoquée par une modification dans l'environnement du travail logiciel et-ou matériel suite à des besoins nouveaux ou changeants. Ce type de maintenance est indispensable afin de s'adapter aux besoins évolutifs du système. Un changement dans la structure du code est un changement logiciel et un changement du système d'exploitation ou du matériel (sur lequel le logiciel est opérationnel) est un changement matériel.
Maintenance perfective : effectuée à la demande des utilisateurs, elle réfère aux modifications apportées au système afin d'augmenter ses performances ou ses fonctionnalités et d'éliminer l'inefficacité du traitement. Elle vise l'amélioration de la qualité du logiciel, la documentation ou autres facteurs de qualité. À titre d'exemple, l'optimisation du code source en vue d'augmenter la rapidité de son exécution. La maintenance perfective coûte de 60 à 70% de l'effort total de la maintenance [28].

Généralement, la maintenance corrective ne constitue qu'un faible pourcentage de l'effort de maintenance. D'autres types de maintenance (non-correctives), visant l'adaptation du système au nouvel environnement du travail (maintenance adaptative) et l'amélioration de ses performances (maintenance perfective) représentent la principale activité de la maintenance. Historiquement, elles comptent pour près de 80% de l'effort total de la maintenance.

2.2.1.2 Processus du changement

Une méthode pour gérer le changement est essentielle pour deux raisons. Elle fournit (i) un canal de communication commun entre le personnel de maintenance, les utilisateurs, le chef de projet et les opérateurs, (ii) un annuaire des changements du système, pour la gestion de projet, l'audit et le contrôle de qualité [57].
Madhavji [54] définit les étapes du processus de changement comme suit :

- identifier le besoin d'apporter le changement à un élément (item) de l'environnement ;
- acquérir le changement adéquat relatif à l'élément ;
- évaluer ou estimer l'impact du changement sur les autres éléments de l'environnement ;
- sélectionner ou construire une méthode pour le processus de changement ;
- faire les changements nécessaires pour tous les éléments, et résoudre les interdépendances de manière satisfaisante ;
- enregistrer les détails des changements pour de futures références ;
- remettre l'élément changé de nouveau dans l'environnement.

Pour bien mener les changements dans un environnement, il est nécessaire de connaître tous les facteurs affectant le changement en question et les conséquences de ce changement.

Bohner décrit un processus détaillé de changement du logiciel qui incorpore l'analyse d'impact [13]. Ce modèle illustre où les impacts du changement peuvent être détectés durant la plupart des activités de changement du logiciel (Figure 2).

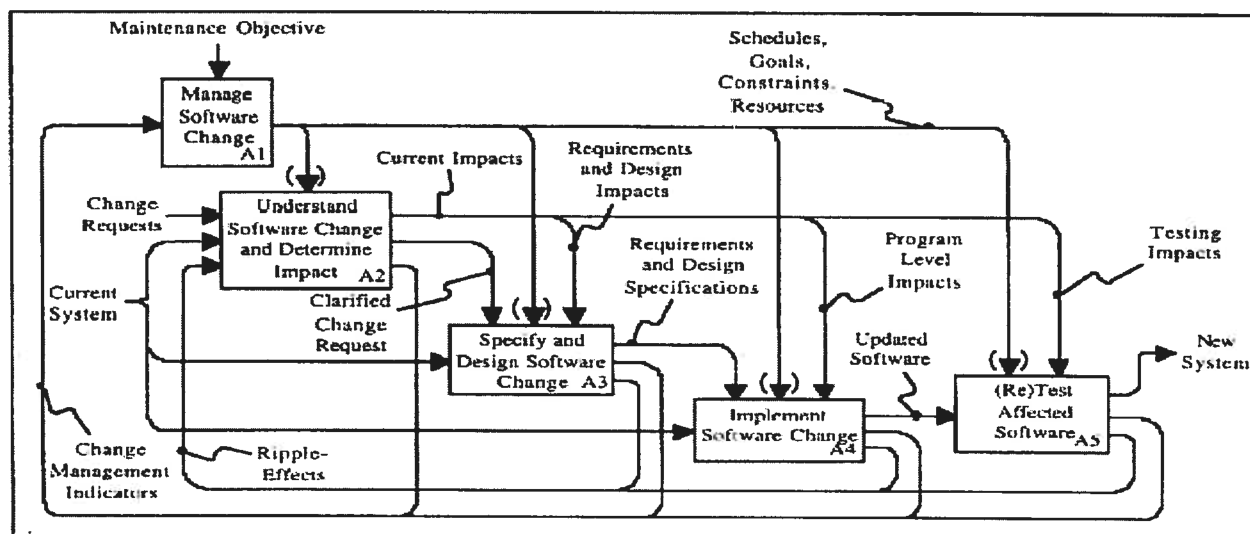

Figure 2 : Processus du changement de logiciel [13]

Ces étapes se présentent comme suit :

A1. Gérer le changement logiciel : étape au cours de laquelle les objectifs de qualité sont établis, les risques des changements sont déterminés et le suivi de la progression du changement, de l'attribution des ressources et de la planification du release sont faites ;

A2. Comprendre le changement et déterminer l'impact : étape qui permet d'identifier les impacts du changement, de clarifier la demande de changement, d'enregistrer les impacts du changement et de déterminer la stabilité du logiciel. L'identification de l'impact représente l'activité centrale qui supporte la plupart des activités du processus de changement. Cette étape a pour objectif de déterminer les impacts du changement du logiciel, de classifier les changements et d'explorer les changements similaires. De plus, elle permet d'identifier les impacts des exigences et de conception, d'analyser les impacts du code source et de déterminer les tests de régression candidats ;

A3. Spécifier et concevoir les changements : étape au cours de laquelle se fait l'analyse des changements, l'examen des changements de l'architecture du logiciel, la dérivation des exigences du changement et la conception du programme de changement ;

A4. Implémenter les changements : c'est une étape itérative qui englobe la détermination des modules à changer, l'identification des expressions du programme à changer, l'application des changements au programme et le test unitaire du logiciel modifié ;

A5. Retester le logiciel affecté : étape qui consiste à générer les cas de tests pour les nouvelles fonctionnalités ou celles modifiées, à mettre à jour la suite de tests, à effectuer les tests d'intégration, à mener le test du système et à réaliser les tests d'acceptation. Elle a pour objectif d'assurer que les modifications effectuées répondent aux nouveaux besoins et que tout le système satisfait les besoins existants.

D'une façon générale, une bonne méthodologie et une parfaite compréhension du système et du changement contribuent largement dans la réduction du temps entre la proposition du changement, son analyse, son implémentation et sa livraison. Par conséquent, moins d'effort est requis pour effectuer le changement. L'implémentation des modifications doit se faire de manière organisée et par priorité. D'une part, les changements qui se croisent doivent être rassemblés de même que les changements similaires afin d'éviter la duplication du travail et l'augmentation de la complexité. D'autre part, dans le but de cerner les effets de bord du changement et son effet de propagation, un seul changement doit être traité à la fois sinon on risque de rendre le système indisponible (surtout qu'on ne sera pas capable de dire quel changement fait échouer les tests).

2.2.2 Analyse d'impact

L'impact est l'effet ou l'influence d'une chose sur une autre. Dans notre étude, l'impact est la conséquence d'un changement. L'analyse d'impact est l'étude de la portée de cet impact. L'analyse d'impact du changement est l'estimation des conséquences d'un changement sur le reste du système. Elle a été pratiquée de différentes manières pendant des années. Cependant, il n'y a pas de consensus sur sa définition [7].

Pfleeger et Bohner définissent l'analyse d'impact du changement comme : *"the evaluation of the many risks associated with the change, including effects on resources, effort, and schedule estimates"* [59]. Arnold et Bohner définissent l'analyse d'impact par : *"impact analysis is the activity of identifying what to modify to accomplish a change or of identifying the potential consequences of a change"*[7]. Turver et Munro la définissent par : *"the assessment of a change, to the source code of a module, on the other modules of the system, it determines the scope of a change and provides a measure of complexity"* [67]. Bohner la définie par : *"impact analysis identifies the consequences or ripple effects of proposed software changes"* [13].

Ces définitions adressent l'analyse d'impact selon différents points de vue. Pfleeger et Bohner dans leur définition mettent en relief l'évaluation de l'impact et s'intéressent à l'aspect gestion, tandis que les autres se concentrent sur l'estimation des conséquences que peut avoir un changement sur le reste du système. L'analyse d'impact est donc une tâche prédictive permettant de planifier les changements et d'identifier les conséquences de leurs propagations avant leurs implémentations.

2.2.3 Avantages de l'analyse d'impact

Quand on est en face d'une nécessité de changement dans un système, il est nécessaire de faire une analyse d'impact. Sans une bonne analyse d'impact du changement, les ingénieurs peuvent faire de petits changements qui peuvent involontairement causer des problèmes majeurs ou avoir des répercussions sur tout le système.

En génie logiciel, cerner les spécifications et les objectifs du projet avant son développement diminue le risque de refaire le travail, de retarder les échéances et de dépasser le budget. Le même principe se présente en analyse d'impact du changement. L'identification des impacts du changement, la planification du changement et l'estimation des ressources requises avant d'implémenter le changement, permettent de diminuer le risque de s'embarquer dans des dépenses coûteuses, inutiles ou imprévisibles.

Lorsqu'un changement se présente, plusieurs solutions sont envisageables pour résoudre le même problème. L'analyse d'impact du changement permet aux gestionnaires de qualifier leurs choix à base d'informations recueillies par l'équipe de maintenance sur la faisabilité technique du changement. L'analyse d'impact est donc un outil d'aide à la décision. En effet, si un changement peut affecter un ensemble de sections disjointes du programme, les gestionnaires peuvent choisir de ne pas l'implémenter ou de l'examiner une autre fois pour une implémentation alternative plus sûre.

L'analyse d'impact du changement permet de diriger les tests de régression (section 2.1.5). Ces derniers jouent un rôle intégral dans la maintenance du logiciel. Les tests de régression appliqués au logiciel modifié permettent d'avoir l'assurance que le code modifié continue à satisfaire les spécifications actuelles et qu'il ne compromet pas le comportement du code non modifié [36]. Suite à une modification, il faut recenser toutes les parties du code qui ont été affectées directement ou indirectement par ce changement. Sans ce recensement, on doit retester tout le

logiciel, ce qui est très coûteux et très long. Ne pas tester suffisamment peut avoir une influence sur la qualité du logiciel.

L'analyse d'impact peut être utilisée pour déduire le coût du changement. Plus la détection du problème est tardive dans le cycle de vie du logiciel, plus le coût de la modification augmente. Plus on comprend l'impact, plus on contrôle le changement. Plus le changement cause d'autres changements, plus le coût augmente. Elle peut aussi être utilisée pour indiquer la partie (ou les parties) critique du code. Une partie dont le fonctionnement dépend d'autres parties du programme est sensible aux modifications apportées à ces dernières.

2.2.4 Travaux de recherche sur l'analyse d'impact des systèmes OO

Plusieurs études ont été menées sous différentes formes à propos de l'analyse d'impact. Han a développé une approche pour calculer l'impact de changement sur les documents de conception et d'implémentation [35]. Kung *et al.* se sont intéressés à l'impact de changement pour les tests de régression [45]. Ils ont classifié les changements et l'impact résultant en se basant sur les relations d'héritage, d'association et d'agrégation. Des algorithmes formels ont été réalisés pour calculer l'ensemble des classes affectées tout en incluant l'effet de propagation (section 2.1.4.2). Une méthode a été proposée pour identifier les classes affectées suite aux changements de structures d'une librairie de classes. Pour chaque structure de changement, l'impact sur les classes résiduelles est calculé pour déterminer si ces classes sont affectées ou non. Afin de calculer l'impact sur tout le système, le concept de firewall (section 2.1.4.3) est utilisé.

Chaumun et Kabaili se sont intéressés à un des aspects de la maintenance qui est la changeabilité [25, 26, 41]. Leur approche consiste à calculer l'impact du changement apporté aux classes d'un système en utilisant le modèle d'impact défini dans [24]. Ce modèle consiste à déterminer l'impact sur les classes directement connectées à la classe changée. Il se base sur deux facteurs :

1. Types de changements pouvant intervenir dans une classe. Par exemple, le changement de type d'une variable a un impact sur toutes les classes qui référencent cette variable, alors que l'ajout d'une variable n'a pas d'impact sur ces classes.
2. Types de liens entre les classes d'un système orienté objet dont l'association (noté par S), l'agrégation (noté par G), l'héritage (noté par H), l'invocation (noté par I),

friendship (noté par F) et local (noté par L). Une partie du système peut être affectée si elle est reliée à la partie changée via les liens qui existent entre elles.

L'exemple suivant illustre cette notion d'impact : lorsqu'on change la portée d'une méthode de publique à protégée, les classes qui invoquent cette méthode seront affectées, à l'exception des classes dérivées. L'impact peut être constitué d'une association des différents types de liens. De ce fait, pour ce changement, noté par *chj,* apporté à une classe, noté par *cli*, l'impact du changement est exprimé par *: Impact (cli, chj)* = S ~H+ G, i.e., les classes affectées sont celles liées avec le lien d'association (S) et qui n'héritent pas de la classe cli (~ signifie non), ou (représenté par +) celles ayant une relation d'agrégation avec la classe cli.

Par ailleurs, ce modèle d'impact a été étendu par Kabaili pour tenir compte de l'effet de propagation (section 2.1.4.2) et des tests de régression (section 2.1.5) afin d'obtenir une meilleure évaluation de la changeabilité du système [42]. Pour déterminer les classes à retester, elle a utilisé une approche inspirée du concept du firewall (section 2.1.4.3) et basée sur les relations de dépendances entre les classes.

Li et Offut analysent l'impact des changements apportés au logiciel OO en prenant en considération l'encapsulation, l'héritage et le polymorphisme [50]. Cette analyse accorde une grande importance aux tests de régression (section 2.1.5) en suggérant les classes et les méthodes qui ont besoin d'être retestées. Ils ont recensé un ensemble de types de changements, analysé leurs caractéristiques et déterminé leurs influences sur les autres parties du système. Un ensemble d'algorithmes est proposé ayant pour objectifs de déterminer l'impact à l'intérieur de la classe changée, l'impact parmi les classes clientes et l'impact parmi les classes dérivées. Combinés, ils permettent d'analyser l'effet de propagation à travers le système. Ces algorithmes calculent la fermeture transitive (section 2.1.4.1) pour chaque classe susceptible d'être affectée par le changement d'un composant. Un ensemble de facteurs qui caractérisent l'impact de la façon dont le changement influence le reste des classes du système est fixé. Un changement peut affecter les classes clientes de la classe changée, et-ou les classes enfants, et-ou la classe changée elle-même ou n'a pas d'effet. Ces facteurs sont :

- Contaminate_all : changement qui affecte les méthodes et les données membres de n'importe quelle classe en relation avec la classe changée (les classes clientes, les sous-classes et la classe changée);
- Contaminate_current : changement qui affecte seulement les données membres et les méthodes de la classe changée;

- Contaminate_client : changement qui ne touche que les classes clientes, i.e., celles qui utilisent la classe changée;
- Contaminate_children : changement qui ne touche que les sous-classes, i.e., celles qui héritent de la classe changée;
- Contaminate_non : changement qui n'affecte aucune classe (ni les classes clients, ni les sous-classes, ni la classe changée elle-même).

Ainsi, lorsqu'une méthode protégée est supprimée, l'impact portera sur la classe elle-même et sur les enfants de cette classe. L'impact de ce changement est noté par : contaminate_current et contaminate_children.

Lindvall, dans son approche de recherche, vise à comprendre les changements les plus communs en C++ pour conduire l'analyse d'impact le plus exactement possible [51]. L'approche est répartie en deux étapes. En premier, une étude empirique a été menée afin de détecter et d'analyser les changements effectués à un code source après que tous les changements aient été effectués et le système livré. La deuxième étude est complémentaire et les développeurs indiquent leurs perceptions sur l'occurrence de certains types de changements. Les résultats de l'étude sont confirmés par les résultats de l'enquête à quelques différences près. En somme, ces résultats permettent de bien comprendre quels sont les éléments (classe, méthode, attribut, protection) qui sont stables et ne changent pas, et quels sont les éléments qui peuvent changer et changent fréquemment.

Antoniol *et al.* proposent une approche pour prédire la taille des changements des systèmes OO en évolution, basée sur l'analyse des classes affectées par la demande de changement [6]. Ils prédisent la taille du changement en terme du nombre de lignes de code (LOC) ajoutées et modifiées. L'approche a été évaluée empiriquement par l'analyse de la relation entre le nombre de LOCs ajoutées/modifiées et le nombre de classes ajoutées/modifiées sur 31 versions d'un système. L'analyse des codes sources a montré que le LOC a augmenté considérablement de la première version à la dernière version. En outre, le nombre de LOC supprimées est généralement faible, comparé au nombre de LOC ajoutées et modifiées et le nombre de LOC ajoutées est plus grand que le nombre de LOC modifiées. Ces observations sur l'évolution du LOC des versions du système suggèrent que pour ce type de système, la plupart des changements affectent les classes au niveau interface (évolution), plutôt que les changements de classes au niveau implémentation (maintenance).

McCrickard *et al.* visent l'impact du changement au niveau architectural. Ils ont étudié les techniques de recouvrement de l'architecture de logiciel quand aucune description d'architecture adéquate n'existe [55]. Des facteurs d'impact ont été établis pour refléter la difficulté des changements :

- Changement de la configuration du fichier ou d'une variable : ces changements ne requièrent pas l'écriture de nouveau code;
- Modification de données, de types ou de structures de données à l'intérieur du code : ces changements ne demandent que des modifications mineures à l'intérieur du code existant. La structure générale et les fonctionnalités du code restent intactes;
- Changement de fonctionnalité du code source pour les modules : ces changements affectent une grande partie du code dans un ou plusieurs modules;
- Changement de l'architecture du logiciel : ces changements nécessitent des ajouts et des suppressions à l'architecture et aux connections des modules.

Pfleeger et Bohner considèrent l'analyse d'impact comme l'activité primaire dans la maintenance du logiciel. Ils ont utilisé une analyse d'impact pour mesurer la stabilité de tout le logiciel avec sa documentation en proposant un nombre de métriques logicielles [59]. Le modèle est basé sur le graphe de traçabilité. Un tel graphe montre les relations parmi le code source, les cas de tests, les documents de conception et les spécifications. Pour chaque "workproduct" (exigences, conception, code, plans de test), la tracabilité verticale exprime les relations parmi les parties du workproduct, et la tracabilité horizontale gère les relations de ses composants par paire de workproduct. La tracabilité (verticale et horizontale) a été représentée en utilisant un graphe orienté (section 2.1.2.3).

Enfin, Arnold et Bohner définissent un modèle conceptuel composé de trois parties pour caractériser et comparer les différentes approches d'analyse d'impact, et ressortir les forces et les faiblesses de chacune d'elles [7]. La première partie du modèle d'analyse d'impact (IA application) examine la méthode utilisée par une approche, pour accomplir l'analyse d'impact. La deuxième partie (IA Parts) s'intéresse au fonctionnement de l'approche, à savoir, ce qu'elle fait, comment elle le fait et les outils utilisés. La dernière (IA effectiveness) s'intéresse à l'efficacité de l'approche d'analyse d'impact.

2.2.5 Démarches d'analyse d'impact

Moreton partage l'analyse d'impact en quatre grandes étapes : (i) détermination de la portée de la requête du changement, (ii) développement des estimations des ressources, (iii) analyse du coût et des bénéfices de la requête du changement, et (iv) le responsable de maintenance conseille les utilisateurs des implications de la requête du changement [57]. Michelle Lee propose un processus d'analyse d'impact plus proche de celui de Moreton, avec des étapes bien ordonnées et des tâches bien spécifiées [48]. Ce processus (Figure 3) est composé de six étapes :

1. Convertir le changement proposé en spécifications de changement.
2. Extraire les informations à partir du code source et les convertir en une représentation interne.
3. Calculer l'impact du changement pour le changement proposé; refaire les trois premières étapes pour d'autres changements.
4. Donner une estimation des ressources basée sur des considérations telles que la taille et la complexité du système.
5. Analyser le coût et les bénéfices des requêtes de changements.
6. Le chef de la maintenance du projet montre aux utilisateurs les implications de la requête du changement, pour décider si le changement doit être effectué.

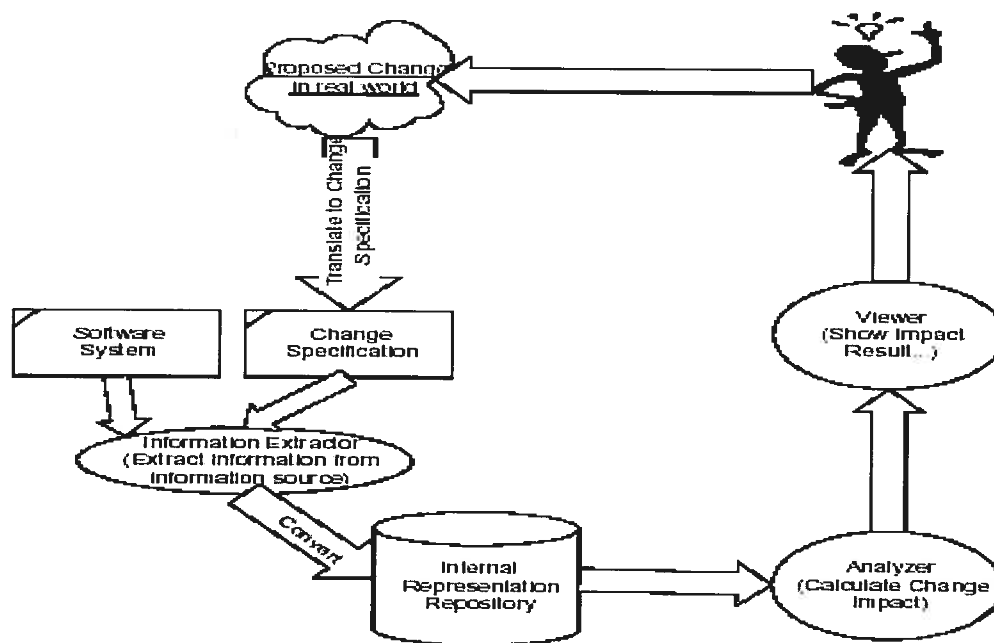

Figure 3 : Processus d'analyse d'impact [48]

2.2.6 Difficultés de l'analyse d'impact

L'analyse d'impact est l'une des parties les plus complexes du processus de changement d'un logiciel. La plupart du temps elle est effectuée quand on commence à se soucier du budget nécessaire pour accomplir le changement. Les langages orientés objets ont été considérés comme des cas difficiles pour l'analyse d'impact [20]. En effet, les caractéristiques de l'approche orientée objet comme le polymorphisme et la liaison dynamique amplifient et compliquent les dépendances entre les méthodes. Les relations complexes entre les classes d'objets rendent difficiles l'anticipation et l'identification de la propagation des effets de changement [45].

Par ailleurs, l'analyse d'impact repose sur des informations statiques connues à la compilation. Le polymorphisme et la liaison dynamique impliquent que les objets peuvent prendre plus d'une forme, qui n'est connue qu'au moment de l'exécution. L'analyse statique est donc insuffisante pour identifier les composants affectés. Faire intervenir le programmeur permet d'apporter plus d'efficacité au processus d'analyse d'impact et d'obtenir des résultats bien précis sur les conséquences du changement. Dans notre recherche, nous nous basons sur l'équipe de développement et son expérience afin d'estimer l'effort dépensé pour faire le changement. Puisque ces personnes sont expérimentées, elles peuvent nous fournir des grandeurs précises.

2.2.7 Conclusion

Nous avons exposé quelques travaux de recherche réalisés dans le domaine d'analyse d'impact du changement et les démarches pour mener l'analyse d'impact et réussir l'implémentation du changement.

L'analyse d'impact représente l'entrée pour effectuer le changement lors de la maintenance. Dans un premier temps, on doit identifier le changement, puis déterminer l'impact du changement sur le système. Cette tâche nécessite la compréhension du logiciel, en utilisant les informations existantes dans les documents de spécification et de conception, et dans le code source. Elle consiste à identifier les parties du système qui peuvent être affectées par le changement et examiner les effets qu'elles peuvent produire à leur tour. La personne chargée de la maintenance peut estimer l'effort nécessaire pour effectuer le changement, puis implémenter le changement. Dans cette phase, il faut être conscient des effets de propagation que peut produire le changement (ripple effect), et finalement retester afin de valider le logiciel. Dans cette étape, il faut trouver parmi les cas de tests existants, ceux nécessaires pour effectuer les tests de régression, et le cas échéant créer de nouveaux tests.

Le facteur principal qui rend la réalisation du changement difficile dans un système orienté objet est les dépendances entre les classes. Un changement affecte la structure, les performances, les fonctionnalités et le comportement du système. La sévérité de cet impact peut dépendre du degré d'interdépendance entre les composants du système.

De façon générale, l'architecture d'un système est vue comme un ensemble de classes interagissant entre elles. La nature de l'interaction entre les classes dépend du type de relations de dépendances existantes entre elles. Plusieurs mesures orientées objets sont utilisées pour évaluer les propriétés architecturales d'un système logiciel telles que le couplage, l'héritage, la cohésion, la complexité, etc. Dans la section qui suit, on présente l'utilité des métriques orientées objets. Quelques métriques de conception en rapport avec l'estimation de l'impact sont présentées. Des travaux de recherche qui ont utilisé les métriques dans le domaine d'analyse d'impact du changement sont aussi cités.

2.3 Métriques orientées objets

L'approche orientée objet est devenue de plus en plus acceptée dans l'industrie du logiciel. Hsia *et al.* ont mené une étude de cas montrant l'effet de l'architecture des systèmes orientés objets sur la maintenance de logiciel [37]. L'étude a montré que la maintenabilité des systèmes orientés objets est meilleure pour ceux possédant des arbres larges, i.e., arbre d'héritage peu profond. Kiran *et al.* ont procédé par comparaison de la maintenance dans le paradigme fonctionnel et le paradigme orienté objet sur des programmes développés par des étudiants [44]. L'expérience a montré que l'effort nécessaire pour effectuer des changements dans le paradigme orienté objet est moins important que celui nécessaire dans le paradigme fonctionnel.

Par ailleurs, avec l'insertion de cette technologie dans l'industrie du logiciel, de nouveaux défis sont crées pour les compagnies qui utilisent les métriques de produits comme outils de gestion, de contrôle et d'amélioration des méthodes de développement et de maintenance de logiciel [11]. C'est pourquoi plusieurs chercheurs se sont dirigés à explorer de nouvelles métriques logicielles spécifiques au paradigme OO [2, 3, 16, 21, 22, 49]. L'apport le plus important est celui de Chidamber et Kemerer. Ils proposent une suite de métriques de conception qu'ils raffinent dans [21, 22, 23]. Cette suite de métriques a été utilisée et expérimentée par *Basili et al.* [11] et Li et Henry [49]. Briand *et al.* ont proposé et validé empiriquement un ensemble de métriques de

couplage au niveau classe [16] et de cohésion [17]. Abreu *et al.* ont défini un ensemble de métriques (MOOD : Metrics for Object-Oriented Design) qui sont empiriquement validées [4].

2.3.1 Quelques métriques de conception orientée objet

Les métriques de conception permettent d'évaluer la structure, les relations et les fonctionnalités des composants d'un système logiciel. Un logiciel peut être analysé à deux niveaux : au niveau système, les caractéristiques externes telles que la hiérarchie des classes et les relations entre elles sont évaluées. Au niveau classe, les caractéristiques internes qui sont évaluées sont les méthodes, les attributs et tous types d'interactions. Parmi les aspects les plus importants dans une conception orientée objet on peut citer la cohésion, le couplage et l'héritage.

2.3.1.1 Métriques d'héritage

L'héritage permet le partage des attributs et des méthodes entre les classes du système. Chidamber et Kemerer [22] identifient les métriques d'héritages suivantes :

- Depth of Inheritance (DIT) : représente la profondeur d'une classe dans l'arbre d'héritage et montre jusqu'à quel point une classe est influencée par ses ancêtres ;
- Number Of Children (NOC) : réfère au nombre de descendants immédiats d'une classe dans la hiérarchie de classes. Elle reflète l'impact d'une classe sur ses descendants.

2.3.1.2 Métriques de couplage

Le couplage est défini comme le degré d'interdépendance entre les composants du système. Deux classes sont dites couplées si une méthode de l'une utilise une méthode ou un attribut de l'autre [22]. Diverses métriques orientées objets permettant de mesurer les différentes facettes du couplage entre classes existent. Chidamber et Kemerer [22] proposent :

- Coupling Between Object (CBO) : représente le nombre de classes avec lesquelles une classe est couplée. Elle reflète le degré d'interdépendance entre les composants du système ;
- Coupling Between Object (CBO') : représente le nombre de classes avec lesquelles une classe est couplée en excluant l'héritage ;
- Response For a Class (RFC) : représente le nombre de méthodes invoquées en réponse à un message. RFC vaut la cardinalité de l'ensemble de réponses d'une classe et mesure la communication entre les classes du système.

Li et Henry [49] proposent :

- Message Passing Coupling (MPC) : représente le nombre de méthodes invoquées; le nombre de messages envoyés par une classe en direction des autres classes du système ;
- Data Abstraction Coupling (DAC) : correspond au nombre d'ADT (Abstract Data Type) définis dans une classe. Une variable peut avoir le type ADT qui représente la définition d'une autre classe.

Briand *et al.*, pour quantifier le couplage entre les classes durant la conception des systèmes orientés objets, ont proposé une suite de métriques de couplage [16]. Puisqu'on s'intéresse aux systèmes développés avec le langage Java, le concept du "friendship" propre au langage C++ n'est pas pris en considération. Cette suite de métriques, présentée dans un tableau en annexe (Annexe 1), concerne l'interaction classe-attribut, classe-méthode et méthode-méthode en tenant compte des descendants, des ancêtres et des autres classes du système. La nomenclature de ces métriques est la suivante :

- La première lettre désigne les types de relation (A pour ancêtres, D pour Descendants et O pour autres). En général, une classe est couplée avec tous ses descendants et ses ancêtres.
- Les deux lettres suivantes indiquent le type d'interaction entre les deux classes :
  - CA (Classe-Attribut) : l'interaction se manifeste à travers les attributs. Une interaction de type CA existe entre les classes A et B, si B a un attribut de type A. Ainsi, si A change, B est affectée via l'attribut de B qui dépend de la classe A.
  - CM (Classe-Méthode) : il existe une interaction de type CM entre les classes A et B, si B a une méthode avec un paramètre de type A. Dans ce cas le couplage est fait via la méthode.
  - MM (Méthode-Méthode) : l'interaction se concrétise à travers les méthodes. Il existe une interaction de type MM entre les classes A et B, si A invoque une méthode de B, ou si une méthode de B est passée comme paramètre à une méthode de A.
- Les deux dernières lettres montrent le sens du couplage (IC, EC) :
  - EC (export coupling) : le couplage d'exportation se présente quand une classe est utilisée par une autre. Notons qu'une classe exporte l'impact à ses descendants et importe l'impact de ses ancêtres.
  - IC (import coupling) : le couplage d'importation se présente quand la classe utilise une autre.

2.3.1.3 Métriques de cohésion

La cohésion reflète le degré d'interaction des méthodes d'une classe. Un logiciel qui obéi au principe de forte cohésion est un logiciel de qualité. Plus la cohésion d'une classe est forte, plus elle est cohérente et constitue une entité entière indissociable et plus simple est sa maintenabilité. Chidamber et Kemerer [22] proposent :

- Lack of Cohesion in Methods (LCOM) : correspond au nombre de paires de méthodes ne partageant aucune variable d'instance, moins le nombre de paires de méthodes partageant des variables d'instances de la classe.

2.3.1.4 Métriques de complexité

Le nombre et la complexité des méthodes sont de bons indicateurs du temps et d'effort nécessaires pour développer et maintenir une classe. Il apparaît logique que plus le nombre de méthodes est grand, plus la classe est complexe. Plus le contrôle de flot de méthodes qu'une classe possède est grand, plus la compréhension est difficile et la maintenance de ses méthodes problématique. Pour mesurer la complexité d'une classe, Chidamber et Kemerer [22] proposent :

- Weighted Methods per Class (WMC) : représente la somme des complexités de toutes les méthodes d'une classe. Si toutes les complexités de toutes les méthodes valent 1, alors WMC équivaut au nombre de méthodes de la classe.

2.3.2 Les métriques orientées objets et l'analyse d'impact du changement

D'après Michelle Lee, les métriques orientées objets pour l'analyse d'impact du changement fournissent des vues numériques de l'effet d'un changement et permettent aux personnes chargées de la maintenance d'évaluer de façon quantitative l'effet des différents changements. Ils peuvent ainsi faire la comparaison entre les différentes alternatives de décisions de maintenance et de conception, aidant l'ingénieur de la maintenance à maîtriser les effets de ses actions sur la structure du logiciel, et donnant une estimation sur l'effort nécessaire pour implémenter le changement [48].

Plusieurs hypothèses ont été formulées sur la possibilité d'existence d'une relation entre les caractéristiques de conception tels le couplage, l'héritage, etc. d'un système et l'impact du changement. Dans le but de valider ces hypothèses, des métriques ont été proposées pour mesurer les caractéristiques de conception, et des études empiriques ont été menées pour tenter

d'établir une corrélation entre ces métriques et l'impact du changement apporté au code source des systèmes OO [20, 25, 26, 41, 42, 48].

Chaumun *et al.* [25], dans leur étude sur la changeabilité des logiciels OO, se sont basés sur le modèle d'impact du changement défini dans [24]. Un seul changement a été retenu pour l'expérience et l'impact a été calculé sur un système industriel. L'objectif était de tester l'influence du WMC sur l'impact du changement. L'expérience a montré que le système absorbe facilement le changement de signature au niveau des méthodes et que les métriques de conception peuvent être un bon indicateur de changeablilité. Par ailleurs, une autre étude a été réalisée en vue de trouver une relation entre la propriété d'architecture relative au couplage et la changeabilité dans les systèmes à objets [26]. Ils ont utilisé un ensemble de neuf métriques de conception. Cinq de Chidamber et Kemerer (22) (WMC, DIT, NOC, CBO et RFC). Les autres sont nouvellement conçues pour détecter la changeabilité. Elles sont au nombre de quatre et dérivées des métriques CBO et NOC ; il s'agit des métriques CBO', NOC*, CBO-U et CBO-IUB décrites en annexe (Annexe 1). L'expérience a été faite sur trois systèmes industriels de tailles différentes. Un ensemble de six changements a été défini et l'impact est calculé : c'est le nombre de classes qui peuvent être affectées. Les résultats ont montré l'existence d'une corrélation significative entre la changeabilité et l'accès à une classe par une autre classe via l'invocation de méthodes ou l'accès aux variables. En outre, la métrique CBO-IUB est un bon indicateur de changeabilité dans le système.

Michelle Lee a développé une nouvelle technique d'analyse pour résoudre le problème de l'analyse d'impact du changement des logiciels OO, plus particulièrement la détermination de l'effet de propagation du changement après son implémentation [48]. Pour ce faire, elle a proposé d'analyser les relations entre les classes d'un système orienté objet (utilisation de graphe de relations) et d'appliquer une technique algorithmique d'analyse du logiciel pour calculer la fermeture transitive de certaines relations entre les classes. Un ensemble de métriques OO d'impact de changement a été conçu et utilisé. Ces métriques, résumées dans le tableau en annexe (Annexe 2), mesurent l'impact dans le système, la classe et les méthodes.

Briand *et al.* [20] ont mené leur étude sur un système commercial en C++, développé et maintenu depuis des années. L'objectif de l'étude est de déterminer si les mesures de couplage permettent de recenser suite à un changement dans une classe l'ensemble des classes affectées. Pour ce faire, ils ont utilisé un ensemble de 18 mesures de couplage dont 4 nouvellement

conçues pour prendre en considération les connections de couplage indirect. Ces dernières peuvent créer l'effet de propagation et par la suite doivent être considérées dans l'analyse d'impact; il s'agit de SIMAS, PIM, PIMAS et INAG résumées dans un tableau en annexe (Annexe 1). L'étude a montré qu'un certain nombre de métriques de couplage en rapport avec l'agrégation et l'invocation sont liées à une probabilité plus élevée de changements communs. En outre, ces métriques de couplage sont de bons indicateurs de l'effet de propagation, et peuvent être utilisées pour classifier les classes en fonction de leurs probabilités de contenir l'effet de propagation.

Kabaili *et al.*, dans l'objectif de trouver une relation entre la cohésion et la changeabilité, ont utilisé deux approches différentes [41]. La première analyse la cohésion en tant qu'indicateur de changeabilité. Les métriques utilisées pour ces fins sont des métriques de cohésion [12] et des métriques de couplage [22] (CBO, RFC, NOC, CBO-IUB, CBO', CBO-U et NOC*). La deuxième approche analyse la capacité de la cohésion à prédire la changeabilité basée sur l'étude de la relation entre les métriques de cohésion et les impacts de changements obtenus par le modèle d'impact sur les systèmes étudiés. L'expérience s'est limitée à six changements. Les résultats des deux approches sont négatifs. Pour s'assurer des résultats, une investigation manuelle des classes supposées être de faible cohésion a été menée. Suite à cette analyse, ils ont conclu que les métriques de cohésion ne peuvent pas être utilisées comme indicateur de changeabilité pour les systèmes à objets.

Enfin, en vue d'étudier la propagation de l'effet du changement, Kabaili *et al.* ont mené une expérience avec le modèle d'impact étendu sur deux systèmes industriels avec neuf changements [42]. Des métriques ont été choisies en utilisant l'approche GQM [10] ; dix d'entre elles de Chidamber et Kemerer [22] (WMC, DIT, NOC, NOC*, CBO, CBO', CBO_IUB, CBO-U, RFC et LCOM), et six de Briand *et al.* [16] (ACAIC, OCMIC, OCAIC, AMMIC, ACMIC et OMMIC). De cette étude, trois conclusions sont établies : (i) la complexité d'une classe prédit sa changeabilité lorsque le changement appliqué au système concerne un changement de méthode ou un changement de classe, (ii) il y a un lien entre la changeabilité et le couplage, et (iii) le nombre de descendants d'une classe parait exercer une influence sur la changeabilité.

2.3.3 Conclusion

Comme cité précédemment dans le chapitre 1 de l'introduction, l'analyse d'impact du changement avec les métriques permet aux gestionnaires de confirmer si le changement répond aux exigences, est conforme à la conception existante, ne dégrade pas la maintenabilité du système existant, et est implémenté de la meilleure manière.

Les métriques permettent de mesurer les différents aspects de couplage, de la cohésion et de l'héritage dans un système orienté objet. Plusieurs études empiriques ont montré la validité de ces métriques comme de bons indicateurs de la maintenance. La validation empirique des métriques permet de démontrer leurs valeurs dans la pratique. Théoriquement, une mesure peut être correcte, alors qu'elle n'a aucune pertinence à l'égard du problème traité. À l'opposé, une mesure peut ne pas être satisfaisante théoriquement, mais peut être utile et intéressante pratiquement. De même, trop mesurer a pour effet une disponibilité d'une grande quantité de données numériques qui sont parfois difficilement gérables. Tandis que peu mesurer, génère peu de données pour mener l'étude et déduire de bonnes conclusions. Dans l'ensemble, la combinaison de plusieurs mesures permet d'avoir un meilleur résultat par rapport à une mesure simple.

Utiliser et valider pratiquement l'utilité des métriques pour l'analyse d'impact du changement était l'un de nos objectifs. Pour estimer l'impact du changement lors de la maintenance des programmes à objets, le chapitre suivant présente notre approche d'analyse d'impact afin d'atteindre ce but.

Chapitre 3 Approche de l'analyse d'impact

Notre travail consiste à estimer l'effort consacré à l'accomplissement du changement en tenant compte de la propagation et à utiliser les métriques orientées objets pour réaliser l'analyse d'impact. Pour ce faire, plusieurs questions se posent :

- Quels sont les outils utilisés pour extraire les métriques ?
- Quelles sont les propriétés du système orienté objet que l'on souhaite mesurer ?
- Quelles sont les métriques de conception orientée objet qui sont de bons repères de l'impact du changement ?
- Quel est le modèle d'impact choisi pour estimer l'impact du changement ?
- Quels sont les changements qui se présentent souvent lors de la maintenance ?
- Quelle est la technique appropriée pour l'évaluation des résultats ?

Les réponses à ces questions bâtissent notre approche d'analyse d'impact du changement illustrée par la figure suivante.

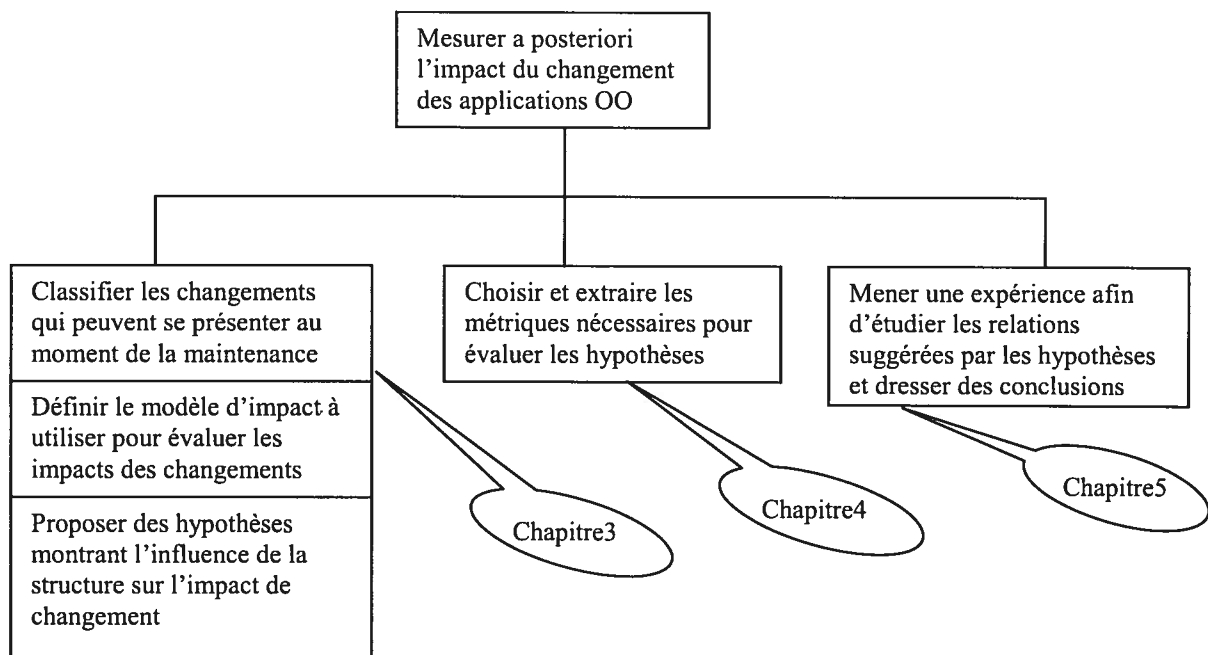

Figure 4 : Approche de l'analyse d'impact

Dans ce chapitre, nous présentons notre approche de l'analyse d'impact qui consiste à définir le modèle d'impact du changement, à proposer des hypothèses montrant l'influence de la structure et les métriques sur l'impact du changement, à introduire les outils d'extraction de métriques (détaillés dans le chapitre 4) et la technique de validation de l'expérience (présentée au chapitre 5). Pour pouvoir estimer l'impact des changements que peut connaître un système orienté objet en maintenance, il faut d'abord identifier ces changements. Dans la section suivante, nous présentons l'étude faite pour classifier ces types de changements.

3.1 Classification des changements

L'identification et la compréhension des changements qui peuvent être apportés aux systèmes orientés objets s'avèrent importantes et fructueuses pour faire l'analyse d'impact. Pour cela, nous avons réalisé un questionnaire visant à rassembler les perceptions des gens, qui assurent la maintenance des systèmes à objets, sur la fréquence de l'occurrence des types de changements, suivi d'une analyse des résultats. Ces deux points sont traités dans les sections suivantes.

3.1.1 Questionnaire des types de changements

Le questionnaire est réparti en deux grandes parties : la première constitue l'ensemble des types de changements et la deuxième, la fréquence de ces derniers. Dans le choix des différents types de changements faisant l'objet du questionnaire, nous nous sommes inspirés des types de changements cités dans la littérature par Chaumun [24], Kung *et al.* [45], Lee [48], et Li et Offut [50]. Ces changements sont groupés par classe, méthode et attribut et se présentent comme suit :

a) Changements au niveau classe :
- Suppression d'une classe.
- Changement d'une classe :
  - Non Abstraite--------------Abstraite.
  - Abstraite --------------------Non Abstraite.
- Ajout d'une classe Abstraite.
- Ajout d'une classe non Abstraite.
- Suppression d'une classe Abstraite
- Suppression d'une classe non Abstraite.
- Ajout d'une super-classe.
- Suppression d'une super-classe.
- Ajout d'une sous-classe.

- Suppression d'une sous-classe.
- Ajout/suppression d'une relation d'héritage/d'agrégation/d'association
- Ajout/suppression d'une classe et de ses relations.
- Ajout d'une classe.
- Ajout/suppression d'une classe indépendante.

b) Changements au niveau méthode :

- Suppression d'une méthode.
- Ajout d'une méthode.
- Changement de la signature :
    - Changement de type de retour.
    - Changement de nom.
    - Changement de la liste des paramètres.
    - Changement du nom d'un paramètre.
- Changement de la portée d'une méthode :
    - Publique----------------Privée.
    - Publique----------------Protégée.
    - Protégée----------------Privée.
    - Protégée----------------Publique.
    - Privée-------------------Publique.
    - Privée-------------------Protégée.
- Changement d'une méthode (static/ non- static) :
    - Static -------------------Non–static.
    - Non-static--------------Static.
- Changement d'implémentation d'une méthode :
    - Ajout/ suppression/ changement d'appel de fonction.
    - Ajout/ suppression/ changement d'une structure de contrôle.
    - Modification de séquence des instructions.
    - Ajout/ suppression d'une séquence d'instructions.
    - Ajout/suppression/ modification d'une variable locale.

c) Changements au niveau attribut :

- Ajout d'un attribut.
- Suppression d'un attribut.
- Changement de la déclaration d'un attribut :
    - Changement de valeur.

  - Changement de type.
  - Changement de nom.
- Changement de la portée d'un attribut :
  - Publique ---------------Privée.
  - Publique ---------------Protégée.
  - Protégée ---------------Privée.
  - Protégée ---------------Publique.
  - Privée ------------------Publique.
  - Privée ------------------Protégée.
- Changement d'un attribut (static/ non- static) :
  - Static ------------------Non–static.
  - Non-static-------------Static.

La fréquence de l'occurrence du changement, notée F, est égale au nombre de fois que le changement se présente dans un système logiciel. Trois types de fréquences sont identifiés : jamais, rare ou souvent. La définition de chaque type est produite de l'expérience menée par Lindvall dans son article [51] :

- jamais / never : le changement ne s'est jamais fait, noté par $F = 0$;
- rare / seldom : le changement s'est présenté moins de cinq fois et au moins une fois, noté par $1 <= F < 5$;
- souvent/ often : le changement s'est présenté plus de cinq fois, noté par $F>=5$.

Le questionnaire sur le recensement des types de changements (Annexe 3) a été présenté à quatre développeurs du CRIM qui assurent le développement et la maintenance de deux systèmes de tailles différentes. Chaque développeur doit reporter pour chaque type de changement, la fréquence relative de l'occurrence parmi les trois types de fréquences cités ci-dessus. Par exemple, pour le type de changement « suppression d'une classe », la question posée est : « pour le système S, combien de fois avez vous supprimé une classe ? » Si la réponse est jamais, alors on coche la case $F = 0$, si rare, on coche la case $1<= F <5$ et si souvent, on coche la case $F >= 5$. (un changement ne peut avoir qu'un seul type de fréquence.)

3.1.2 Analyse des résultats

Étant recueillis, les résultats du questionnaire sont organisés par types de changements aux niveaux classe, méthode et attribut, dans trois tableaux présentés en annexe (Annexe 4). On reporte le nombre de sujets qui ont choisi pour un type de changement l'une des fréquences. Par exemple, pour le type de changement « suppression d'une méthode », les quatre sujets ont

reporté que ce changement se présente rarement. Ensuite, une classification de ces changements est faite selon les types de changements qui se présentent jamais, rarement ou souvent. Pour cela, nous avons utilisé les définitions fournies par Lindvall [51] concernant ces trois groupes.
Ainsi, pour qu'un changement figure parmi le groupe des changements qui se présentent :

- **jamais**, il faut que le nombre de développeurs ayant répondu que ce changement a l'occurrence jamais soit supérieur au nombre de développeurs ayant répondu par rare, qui est supérieur ou égal au nombre de développeurs ayant répondu par souvent.

  $$\text{Jamais} > \text{Rare} \geq \text{Souvent}$$

- **rare,** il faut que le nombre de développeurs ayant répondu que ce changement a l'occurrence rare soit supérieur ou égal au nombre de développeurs ayant répondu par souvent et soit supérieur ou égal au nombre de développeurs ayant répondu par jamais.

  $$\text{Rare} \geq \text{Souvent} \ \& \ \text{Rare} \geq \text{Jamais}$$

- **souvent**, il faut que le nombre de développeurs ayant répondu que ce changement a l'occurrence souvent soit supérieur au nombre de développeurs ayant répondu par rare, qui est supérieur ou égal au nombre de développeurs ayant répondu par jamais.

  $$\text{Souvent} > \text{Rare} \geq \text{Jamais}$$

Les résultats de la classification des différents types de changements selon les trois groupes précités sont présentés dans les sections suivantes.

3.1.2.1 Changements dont l'occurrence est "jamais"

1. Changement d'une classe Non Abstraite-------------Abstraite.
2. Changement d'une classe Abstraite ------------------Non Abstraite.
3. Ajout d'une classe Abstraite.
4. Suppression d'une classe Abstraite.
5. Ajout d'une super-classe.
6. Suppression d'une super-classe.
7. Changement de la portée d'un attribut :
   a. Protégée---------------Privée.
   b. Protégée---------------Publique.
   c. Privée------------------Protégée.
   d. Publique---------------Privée.
   e. Publique---------------Protégée.
8. Changement d'un attribut static------ Non static.

9. Changement de la portée d'une méthode :
   a. Protégée---------------Privée.
   b. Protégée---------------Publique.
10. Changement d'une méthode static ------------Non static.
11. Changement d'une méthode non static--------Static.
12. Changement d'un attribut non static-----------Static.
13. Changement de la valeur d'un attribut.
14. Ajout d'une sous-classe.
15. Suppression d'une sous-classe.
16. Ajout/suppression d'une classe et de ses relations.

3.1.2.2 Changements dont l'occurrence est "rare"

1. Ajout d'une classe.
2. Ajout d'une classe non abstraite.
3. Changement de type d'un attribut.
4. Changement de nom d'un attribut.
5. Suppression d'une méthode.
6. Changement du nom d'une méthode.
7. Changement d'implémentation d'une méthode, modification de séquence des instructions.
8. Changement de la portée d'une méthode :
   a. Privée-----------------Publique.
   b. Publique--------------Privée.
   c. Publique--------------Protégée.
   d. Privée-----------------Protégée.
9. Changement de la signature d'une méthode :
   a. Changement du nom de paramètre.
   b. Changement de la liste des paramètres.
   c. Changement de type de retour.
10. Ajout d'un attribut.
11. Suppression d'une classe.
12. Ajout/suppression d'une relation d'héritage, d'agrégation ou d'association.
13. Suppression d'un attribut.

3.1.2.3 Changements dont l'occurrence est "souvent"

1. Ajout d'une méthode.
2. Changement d'implémentation d'une méthode :
   a. Ajout/suppression/changement d'appel de fonction.
   b. Ajout/suppression/changement d'une structure de contrôle.
   c. Ajout/suppression d'une séquence d'instructions.
   d. Ajout/suppression/modification d'une variable locale.

3.1.2.4 Synthèse des résultats

Les principaux résultats de la classification des changements se résument comme suit :

- Les classes et l'arbre d'héritage présentent peu de changement. Avec l'évolution du système, des méthodes et des attributs sont ajoutés et parfois même des classes ;
- Les classes, les méthodes et les attributs sont rarement supprimés ;
- Les noms d'attributs et de méthodes sont relativement stables ainsi que la signature de méthode (noms des paramètres, liste des paramètres et type de retour) ;
- La portée des attributs est stable et ne présente pas de changement, de même que les méthodes protégées ;
- Les méthodes sont beaucoup ajoutées ;
- Les implémentations de méthodes présentent beaucoup de changements ainsi que les appels de fonctions.

De cette classification, on constate que les super-classes et les sous-classes ne sont jamais supprimées. Ceci implique que la structure du système, notamment l'arbre d'héritage soit souvent stable. En outre, la visibilité des attributs ne présente pas de modification, de même pour les méthodes qui ont une portée protégée, autrement dit, celles visibles pour la classe et ses descendants.

Par ailleurs, des classes sont rarement ajoutées puisque l'ajout d'une classe nécessite possiblement l'ajout de nouvelles relations d'héritage et-ou d'utilisation. De même, la suppression de classes se présente rarement. Si le cas se présente, les gens qui assurent la maintenance doivent changer toutes les références à la classe en question pour que le système ne perde pas ses fonctionnalités. Les noms de méthodes et d'attributs sont relativement stables. La signature de méthode est aussi stable, en particulier les noms des paramètres, la liste des paramètres et le type de retour. Ceci présente un avantage, vu qu'on n'a pas besoin de changer

les appels de fonctions existants. Le changement de la liste de paramètres nécessite un examen de toutes les références à cette méthode pour s'assurer de leur validité. La suppression de méthodes et d'attributs fait aussi partie de cette catégorie évitant ainsi les changements de toutes les références à ces méthodes et attributs.

La catégorie des changements qui se produisent plus souvent englobe (i) l'ajout de méthodes afin de remplir les nouvelles fonctionnalités et satisfaire les besoins changeants des utilisateurs, (ii) le changement de l'implémentation d'une méthode via le changement de traitement (ajout/suppression de séquences d'instructions et de structures de contrôle), et (iii) l'ajout de nouveaux appels de fonctions. En effet, avec l'ajout d'une méthode, de nouveaux appels de fonctions doivent être ajoutés entraînant ainsi le changement des implémentations des méthodes qui la référencient.

3.1.3 Conclusion

Dans cette section, nous avons mené une expérience sur deux systèmes en maintenance avec quatre développeurs afin de classifier les changements qui se présentent lors de la maintenance des systèmes orientés objets. Lindvall [51] a mené une expérience similaire sur un seul système en maintenance avec huit développeurs. En général, le résultat de notre expérience et celle de Lindvall convergent dans les cas suivants :

- l'ajout et le changement d'implémentation de méthode sont parmi les changements qui se présentent le plus ;
- l'ajout de classes est le type de changement dont l'occurrence est rare.

Ils sont proches dans d'autres cas. La suppression de classes, de méthodes, d'attributs et le changement de nom de méthode ou d'attribut se présentent rarement dans notre expérience, alors que pour Lindvall, ils existent parmi la catégorie des changements qui ne se présentent jamais. Par ailleurs, Antoniol *et al.*, dans leur étude, ont conclu qu'en général, aucune classe n'est supprimée, en passant d'une version à la suivante, et peu de classes sont ajoutées et la plupart des classes ne sont pas changées [6]. Leur expérience portait sur un système en évolution plutôt qu'un système en maintenance.

La classification des changements selon les occurrences : jamais, rare ou souvent facilite le choix de la liste des changements que l'on désire évaluer avec le modèle d'impact du changement défini dans la section suivante.

3.2 Définition du modèle d'impact de changement

Dans le chapitre 2 sur l'état de l'art, la section 2.2.4 a présenté un ensemble de travaux de recherche effectués dans le domaine de l'analyse d'impact. Ceci a donné naissance à une diversité d'approches de l'analyse d'impact du changement qui ont profondément inspiré notre conception des modèles d'impacts présentés au fil de cette section. Avant de présenter nos modèles d'impact, une introduction de nouveaux concepts et définitions s'avère nécessaire.

3.2.1 Nouveaux concepts/définitions

3.2.1.1 Définition d'impact

Tel que déjà avancé, l'impact est l'effet d'un changement apporté à une partie du système et qui peut provoquer un dysfonctionnement ou un changement de comportement dans les autres parties du système. Par conséquent, la modification de ces parties s'avère cruciale afin que le logiciel continue à satisfaire les spécifications actuelles et nouvelles.

L'impact d'un changement apporté à une classe sur le reste du système est déterminé avant, au moment et après l'implémentation du changement, i.e., lors de l'analyse du code source, de l'implémentation du changement et lors de l'exécution du système. Nous définissons l'impact par le degré de difficulté ressentie pour achever le changement. Cette difficulté est soit nulle, faible, moyenne ou grande.

3.2.1.2 Propagation d'impact via les relations directes

Une classe du système est affectée si elle est liée à la classe changée par une des relations directes. La propagation d'impact via ces relations est illustrée comme suit :

a) Agrégation : L'exemple suivant montre la propagation d'impact via la relation d'agrégation :

- La classe Class_ A est une agrégation de Class_B. Puisque la méthode mA() de Class_A appelle mB() de Class_B, alors Class_B peut avoir un impact sur Class_A.

```
class Class_A {
Class_B b;
void mA(){ b.mB(); }}
```

- la classe Class_B est une agrégation de Class_C. Vu que la méthode mB () de Class_B appelle mC() de Class_C, alors Class_C peut avoir un impact sur Class_B.

```
class Class_B {
Class_C c;
void mB(){ c.mC(); }}
```

- En conséquence, la classe Class_C peut avoir un impact sur Class_A.

```
class Class_C{
...
void mC(){...}}
```

De cette manière, un changement dans la méthode mC () de la classe Class_C affectera la classe Class_B, qui affectera à son tour la classe Class_A.

b) Utilisation : L'exemple préalablement cité pour l'agrégation est valable aussi pour la relation d'utilisation. Ainsi, si Class_A utilise Class_B et Class_B utilise Class_C, alors Class_C peut avoir un impact sur Class_A. La relation d'utilisation correspond au concept de couplage entre les classes qui, plus la classe est couplée à d'autres classes, plus l'impact du changement est grand et plus l'effort de la maintenance augmente. En effet, si une classe est affectée, ses clients le seront aussi.

c) Héritage : La relation d'héritage est transitive : si Class_A hérite de Class_B et class_B hérite de Class_C, alors Class_A hérite de Class_C. Par conséquent, un changement dans Class_C peut avoir un impact sur Class_A. Le concept d'héritage augmente le couplage entre les super-classes et les classes dérivées. Ainsi, si la classe dérivée étend ses services en se basant sur les services offerts par la classe de base, un changement dans la méthode de la classe de base peut se propager dans l'arbre d'héritage et affecter l'ensemble des descendants de cette classe.

3.2.1.3 Catégories d'effets de changements

Les conséquences d'un changement peuvent être à l'intérieur de la classe changée, et-ou dans les autres classes liées à cette classe (Figure 5). On peut donc dégager deux types d'effets de changements :

- effet interne : les conséquences du changement se limitent à la classe changée ;
- effet externe : les conséquences du changement touchent les classes liées à la classe changée.

Par ailleurs, l'effet externe d'un changement peut prendre deux formes :

- effet direct : causé par la relation de dépendance directe entre la classe changée et la classe affectée ;
- effet indirect : causé par l'existence d'une chaîne de relations entre une série de classes (débute par la classe changée et finit par la dernière classe dans la chaîne de relations).

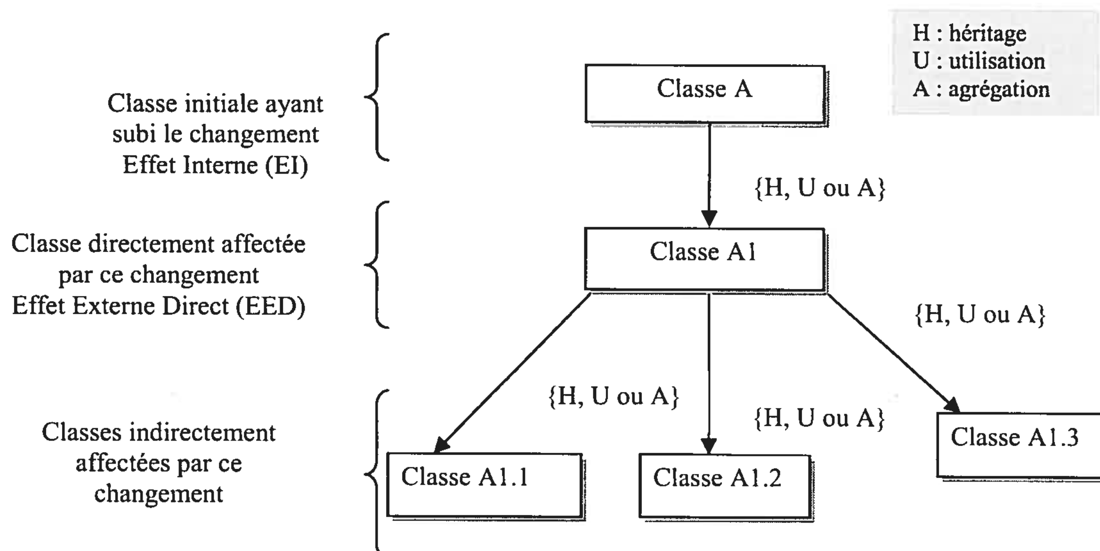

Figure 5 : Catégories d'effets de changements

En résumé, les trois types d'effets qu'une classe sujette au changement peut avoir sur le système dans sa globalité sont :

- Effet Interne (EI) : les conséquences du changement sont à l'intérieur de la classe changée ;
- Effet Externe Direct (EED) : les conséquences du changement concernent les classes directement liées avec la classe changée ;
- Effet Externe Indirect (EEI) : les conséquences du changement se propagent pour atteindre les autres classes (ripple effect), obligeant le programmeur à modifier ou retester ces classes (tests de régression).

3.2.2 Modèles d'impact du changement

Divers chercheurs ont étudié l'impact du changement de différentes manières (section 2.2.4) dont les plus importantes [24, 42, 50, 55] ont influencé notre conception des modèles d'impact. Dans cette section, nous proposons trois modèles d'impact du changement : statique quantitatif, fonctionnel et statique qualitatif à partir desquels nous avons choisi le modèle d'impact utile

pour notre travail. Chaque modèle est constitué de quatre niveaux qui varient selon leurs valeurs d'impact nulle, faible, moyenne ou grande.

Modèle statique quantitatif : l'objectif de ce modèle est de compter le nombre de classes touchées par un changement. Il est composé de quatre types d'impact :

- Impact Nul : aucune classe n'est affectée par le changement ;
- Impact Faible : une classe est affectée par le changement ;
- Impact Moyen : moins de 5 classes et plus d'une classe sont affectées par le changement;
- Impact Grand : plus de 5 classes sont affectées par le changement.

Ce modèle représente la manière la plus facile de calculer l'impact du changement, i.e., de compter le nombre de classes affectées par le changement. Il est plus simple, sauf que dans notre cas nous ne pouvons avoir cette information puisque le report du nombre de classes s'avère difficile (pas d'historique des changements).

Modèle fonctionnel : ce modèle s'intéresse à l'impact du changement dans les fonctionnalités et l'architecture du système. La description des différents niveaux d'impact est comme suit :

- Impact Nul : changement n'affectant ni les fonctionnalités ni l'architecture du système ;
- Impact Faible : changement affectant les fonctionnalités de la classe changée. Ce changement nécessite une faible modification dans le code source ;
- Impact Moyen : changement entraînant une modification dans les fonctionnalités d'une ou de plusieurs autres classes ;
- Impact Grand : changement entraînant des ajouts ou des suppressions dans l'architecture des classes et leurs connexions. Ceci implique une ré implémentation d'une ou de plusieurs classes.

Ce modèle met l'emphase sur les changements qui touchent la conception proprement dite d'un système, principalement les fonctionnalités (les spécifications du système) et l'architecture (les relations entre les composants du système). Étant donné que notre intérêt porte sur la détermination de l'impact à travers les classes d'un système orienté objet, ce modèle reste non applicable dans notre projet.

Modèle statique qualitatif : l'objectif visé par ce modèle est de déterminer l'étendue de l'impact sur tout le système. Ses quatre niveaux d'impact se présentent comme suit :

- Impact Nul (IN) : le changement n'a aucun impact ni localement ni sur le reste du système (ni effet interne, ni effet externe direct, ni effet externe indirect) ;
- Impact Faible (IF) : le changement a un impact local (effet interne) et pas d'impact sur les autres classes du système (pas d'effet externe ni direct ni indirect) ;
- Impact Moyen (IM) : le changement peut avoir ou ne pas avoir un impact sur la classe changée. Cependant, il a un effet sur les classes reliées directement avec la classe changée (effet externe direct) ;
- Impact Grand (IG) : le changement peut avoir ou ne pas avoir un impact dans la classe changée. Cependant, il a un impact sur le reste du système (effet externe direct et effet externe indirect).

Ce modèle d'impact met l'emphase sur l'étendue de l'impact du changement. Il permet d'identifier l'impact sur tout le système en tenant compte de l'effet de propagation de l'impact du changement à travers les classes du système. En effet, un changement apporté à une classe du système peut avoir un effet dans la classe elle-même (EI), et-ou dans les classes directement connectées à la classe changée (EED), et-ou dans les classes qui sont indirectement connectées à la classe changée (EEI). La détermination de ces effets se base sur les relations de dépendances directes entre les classes telles que l'héritage, l'utilisation et l'agrégation, et sur les dépendances indirectes.

Ce dernier modèle d'impact répond à notre but, puisqu'il permet de déterminer l'impact du changement à travers tout le système. Pour estimer l'impact, nous nous basons sur la perception de l'équipe de développement et son expérience pour estimer l'effort que dépense les développeurs pour faire un changement.

3.3 Hypothèses sur l'impact du changement

L'analyse d'impact du changement peut être influencée par plusieurs facteurs comme l'expérience des développeurs, la compréhension du logiciel, la formulation exacte du type de changement et la structure du logiciel. Cependant, ces facteurs ne sont pas toujours tous disponibles, ce qui a une grande influence sur l'estimation du coût du changement.

Étudier l'influence des propriétés de conception orientée objet sur l'impact du changement, en particulier le couplage et l'héritage est l'un des buts de ce projet. Pour cela, des propositions d'hypothèses sont établies et présentées dans les sections suivantes.

3.3.1 Héritage et impact du changement

Un système logiciel est composé d'un ensemble de classes qui forment son architecture, et par l'identification de cette architecture, on simplifie la compréhension du changement, ainsi que l'identification de l'impact.

L'héritage est une relation d'hiérarchie entre les classes d'un système dans laquelle une classe partage ses propriétés avec une ou plusieurs classes. En effet, une sous-classe peut accéder ou faire référence à un attribut de sa super-classe et une méthode dans la sous-classe peut invoquer une autre de sa super-classe. Avec l'héritage au lieu de rassembler les attributs et les méthodes au sein d'une classe, ils sont dispersés sur plusieurs sous-classes et donc augmente leurs interdépendances.

Il apparaît logique que plus une classe est profonde dans l'arbre d'héritage, plus elle peut accéder aux propriétés de ses super-classes. Un changement dans une des super-classes entraîne une propagation d'effet du changement dans les sous-classes. D'un autre côté, plus une classe a d'enfants directs, plus les autres classes peuvent être affectées à cause de l'héritage. À titre d'exemple, si une classe possède plusieurs sous-classes qui dépendent de quelques méthodes ou variables d'instances définies dans la super-classe, un changement dans ces méthodes ou variables peut affecter les sous-classes, ce qui rend la maintenance d'une classe difficile [49]. L'héritage n'isole donc pas les effets de changements apportés, mais au contraire il les réplique. Notre proposition d'hypothèse est comme suit :

Hypothèse 1 : *la position de la classe dans l'arbre d'héritage d'un système orienté objet affecte l'impact du changement.*

3.3.2 Couplage et impact du changement

Le couplage a été introduit initialement par Stevens *et al.* dans le contexte de la conception structurée [65]. Il mesure la force de l'interconnexion entre les modules d'un système modulaire. En orienté objet, un module représente une classe. Un logiciel de bonne qualité doit obéir au principe de faible couplage [16, 18, 22, 52].

Un couplage faible facilite la maintenance vu que les dépendances entre les classes du système sont minimes. Plus encore, il permet de minimiser les chemins de propagation du changement par lesquels l'impact du changement initial peut se propager vers les autres classes du système. Ainsi, plus les dépendances sont faibles, moins important est l'effet de propagation du changement, tandis qu'un couplage fort entre deux classes entraîne plus de dépendances. En conséquence, une modification d'une classe peut entraîner une modification des classes

auxquelles elle est liée. Plus encore, la compréhension d'une classe indépendamment de son environnement est plus difficile à cause du flot important de dépendances.

Si le nombre de méthodes d'une classe qui peuvent être invoquées en réponse à un message est grand, la compréhension de cette classe devient plus compliquée. Ce qui rend l'impact du changement apporté à cette classe important. Plus le couplage d'exportation (Export Coupling) d'une classe C est élevé, plus grand est l'impact du changement apporté à C sur les autres classes. Ces dernières dépendent de façon critique de la conception de la classe C. Plus le couplage d'importation (Import Coupling) d'une classe C est élevé, plus grand est l'impact du changement dans les autres classes sur C elle-même. Donc, la classe C dépend de façon critique de ces classes, et par la suite, la compréhension de C s'avère difficile [16]. On propose l'hypothèse suivante :

Hypothèse2 : *le degré d'interdépendance entre les classes affecte l'impact du changement.*

3.4 Extraction des métriques

Dans notre étude, on s'est intéressé à l'analyse d'impact du changement apporté au code source des systèmes développés avec le langage Java. Car Java (langage de programmation orienté objet) est devenu le langage de choix dans les industries, spécialement pour les systèmes distribués et les applications Web, pour sa portabilité et son indépendance vis à vis des plates-formes.

Notre objectif était d'extraire des métriques capturant les caractéristiques importantes comme le couplage et l'héritage et permettant de valider les hypothèses formulées dans la section précédente. Pour cela, nous avons exploré un ensemble d'outils d'extraction de métriques et nous avons choisi d'utiliser l'outil BOAP (Boite à Outils d'Analyse de Programmes) développé par le CRIM. C'est un ensemble d'outils logiciels intégrés qui permet d'évaluer la qualité des systèmes logiciels. En outre, il est doté d'un module d'extraction des métriques à partir du code source d'une application orientée objet. Cependant, face à la rareté des métriques de couplage implémentées dans BOAP et compte tenu du problème de modèle présent dans la version courante de BOAP, limitant l'implémentation de celles faisant référence aux méthodes, nous avons développé un autre outil sous la plate-forme Eclipse (plate-forme d'intégration d'outils). Il permet d'analyser un code source Java et d'extraire des métriques.

Ainsi l'extraction des métriques se fera avec deux outils : module générique d'extraction des métriques de BOAP et l'outil que l'on a développé pour pallier aux insuffisances du premier. Ces deux outils sont présentés au chapitre 4 avec les métriques que l'on peut extraire.

3.5 Estimation de l'impact de changement par les modèles prédictifs

Divers auteurs ont utilisé de multiples techniques d'analyse dans leurs travaux de recherche. Li et Henry ont utilisé la régression linéaire dans le but de créer des modèles prédictifs qui relient les métriques orientées objets et la maintenabilité [49]. Le principe de la régression linéaire est de sélectionner à partir d'un ensemble de variables indépendantes X, un sous-ensemble de variables qui pourra expliquer les variances dans une variable dépendante. Briand *et al.* dans [19], afin de relier la complexité de la conception et l'architecture du système avec les composants ayant une fréquence de fautes grande/faible, ont utilisé une technique de classification usuellement utilisée quand la variable dépendante est de nature binaire. On parle de la régression logistique, appliquée aussi par Briand *et al.* dans [20] pour déterminer les mesures qui sont des indicateurs utiles de l'effet de propagation. Toujours avec la même technique, Basili *et al.* [11] ont effectué une validation empirique des métriques définies dans [22] en ce qui concerne leur habilité à prédire la probabilité de détecter les classes qui présentent des fautes. Antoniol *et al.* [6] ont considéré plusieurs techniques de régression afin de modéliser les relations entre la taille des changements et les métriques concernant l'évolution des entités orientées objets. Chaumun *et al.* dans [25], pour trouver une corrélation entre l'impact du changement et les métriques de conception orientée objet, ont utilisé l'approche ANOVA, acronyme de "ANalysis Of VAriance". C'est une technique statistique dont l'objectif est d'étudier l'influence d'un ou de plusieurs facteurs sur une variable quantitative.

D'autres travaux de recherche ont utilisé des algorithmes d'apprentissage. Ces derniers permettent à partir de données fournies en entrée de construire des modèles. Almeida *et al.* ont effectué une étude empirique dans laquelle ils étudient différentes techniques d'algorithmes d'apprentissage et leurs capacités de générer des modèles d'évaluation de la maintenance corrective [5]. Lounis *et al.* [52] ont proposé un ensemble de mesures de couplage pour quantifier le niveau de couplage dans un système logiciel modulaire. Ces métriques ont été empiriquement validées en utilisant un modèle prédictif basé sur des algorithmes d'apprentissage.

Nous avons choisi d'utiliser les techniques d'algorithmes d'apprentissage pour valider nos hypothèses. Le plus grand avantage des algorithmes d'apprentissage réside dans l'interprétation des résultats. De plus, ils produisent des modèles prédictifs de bonne qualité, comparables aux modèles basés sur l'analyse statistique. Un ensemble d'algorithmes, présentés dans le chapitre 5, sera utilisé dans le but de construire le meilleur modèle possible.

3.6 Conclusion

Le présent chapitre propose notre approche d'analyse d'impact qui inclut les différents types de changements appliqués à un système orienté objet. En effet, une expérience visant à identifier la liste des changements et les fréquences de leurs occurrences a été menée sur deux systèmes de tailles différentes. Pour ce faire, nous avons conçu un questionnaire dans le but de rassembler les perceptions des développeurs des deux systèmes de l'étude sur l'occurrence des changements. L'analyse du résultat du questionnaire permet de classifier les changements selon trois types de fréquences : jamais, rare ou souvent.

L'impact est exprimé par le degré de difficulté ressentie pour achever un changement. Le modèle statique qualitatif d'impact du changement est choisi ; son objectif est de déterminer la propagation de l'impact parmi les classes du système. Dans le but d'étudier l'influence des propriétés de conception orientée objet sur l'impact du changement, des hypothèses ont été proposées concernant le couplage et l'héritage.

Le modèle d'impact choisi sera utilisé, dans le chapitre 5 de l'expérience, pour estimer l'impact d'un ensemble de changements. D'autre part, pour trouver les métriques orientées objets indicatrices de l'impact de changement, deux outils d'extraction de métriques sont utilisés. Une description de ces outils et des métriques que l'on peut extraire fait l'objet du chapitre suivant.

Chapitre 4 Extraction des Métriques

Notre objectif consiste à mesurer l'impact du changement dans les systèmes OO. Après avoir présenté au chapitre 3 l'approche d'analyse d'impact, le présent chapitre consiste à traiter l'extraction des métriques. Nous avons exploré un ensemble d'outils pour ces fins, notre choix consiste à utiliser BOAP. Ce dernier permet l'extraction de divers types de métriques. Cependant, vu la rareté de celles nécessaires pour notre travail, nous avons procédé à l'extension de son module d'extraction des métriques via l'implémentation de nouvelles métriques. Vu que l'implémentation de certaines d'entre elles ne peut être faite sur la version actuelle de la plate-forme BOAP, notamment celles répondant à nos attentes, nous avons développé un nouvel outil qui permet d'analyser un code Java et d'extraire ces métriques. Les sections suivantes présentent ces deux outils et les travaux que nous avons réalisés.

4.1 Extraction des métriques avec BOAP

BOAP, Boite à Outils pour l'Analyse des Programmes développés au CRIM, est un ensemble d'outils logiciels intégrés qui permet à un expert d'évaluer rapidement le niveau de qualité d'un logiciel selon les objectifs visés [29]. Ces outils sont centrés autour de l'analyse statique du code source et de la compréhension de programmes à travers plusieurs abstractions du code source d'un système orienté objet.

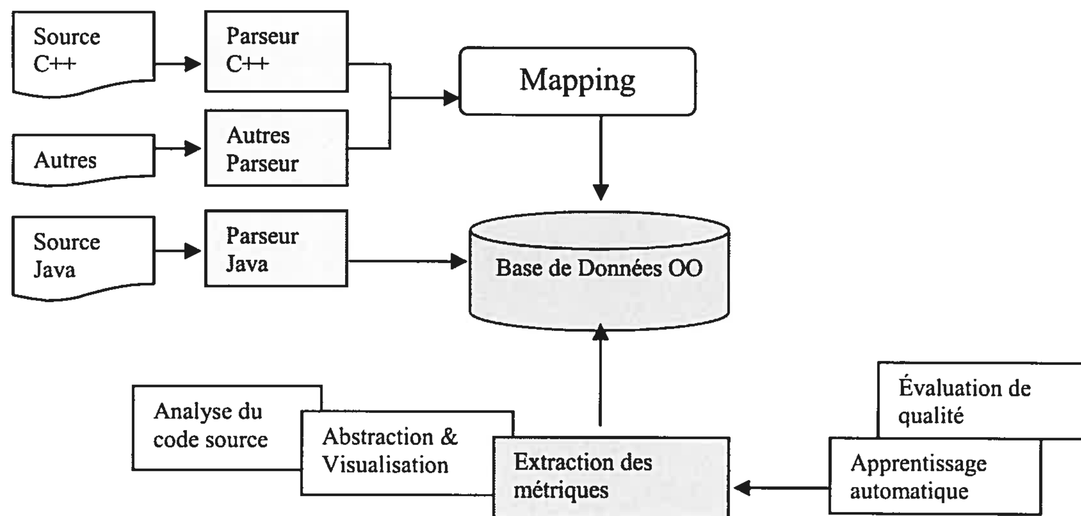

Figure 6 : Architecture générale de BOAP

L'architecture de BOAP permet l'extraction de code à travers le module chargé de l'analyse syntaxique des systèmes. Durant l'analyse (avec Datrix C++, Java ou autres) d'un système, la génération de la représentation du code source est effectuée et sauvegardée dans une base de données orientée objet. L'implémentation de cette dernière est basée sur *ObjectStore*, qui permet la persistance des objets sur lesquels vont agir tous les autres modules de BOAP dont les modules de visualisation, d'abstraction, d'analyse et d'évaluation de la qualité et d'extraction de métriques.

4.1.1 Analyse du module de métriques de BOAP

Dans cette section, nous procédons à l'analyse du module de métriques de BOAP afin de faire ressortir les insuffisances en matière de métriques. Pour cela, nous présentons le module de métriques de BOAP, suivi d'une analyse et discussion. Suite au résultat de l'analyse, nous concluons par un tableau regroupant la liste des métriques à implémenter.

4.1.1.1 Présentation du module de métriques

BOAP dispose d'un module d'extraction et d'affichage de métriques. Ces métriques sont extraites du code source d'une application orientée objet. Il s'agit de métriques d'héritage, de complexité, de cohésion et de couplage, et couvrent les différents aspects mesurables d'une application tels que le système, le module, le fichier, la classe, la méthode et la routine. En outre, ce module est doté de trois outils facilitant la tâche des utilisateurs du système. Le premier permet de calculer et de visualiser les métriques. Le deuxième offre une interface graphique par laquelle de nouvelles métriques peuvent être définies de manière interactive. En effet, ces métriques sont déduites de celles calculées par le système en utilisant des opérateurs arithmétiques et des fonctions mathématiques tels que la moyenne, la somme, etc. Le dernier outil, graphique, sert à configurer et personnaliser les paramètres des métriques. L'utilisateur spécifie les informations qui doivent être prises en considération dans le calcul des métriques.

Afin de comprendre ce module et de pouvoir implémenter et intégrer de nouvelles métriques au module déjà existant, une compréhension de son architecture générale s'avère importante. Dans cette section, nous allons décrire les différentes classes représentant le package des métriques du système BOAP.

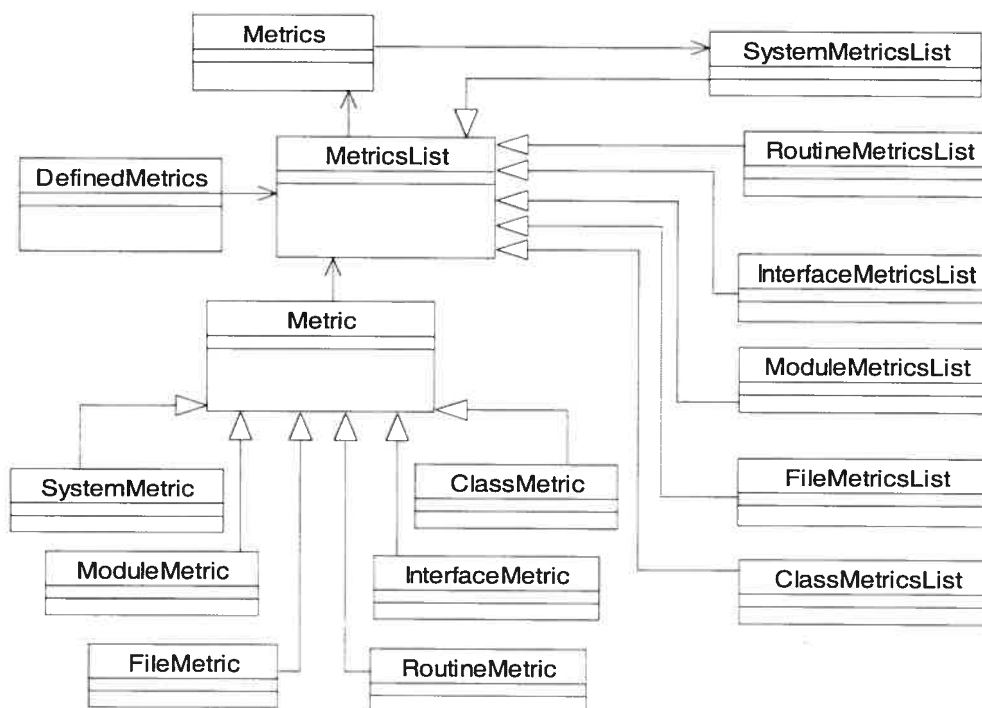

Figure 7 : Diagramme de classes du package de métriques de BOAP

Les métriques de BOAP sont réparties sur différents aspects d'un système orienté objet dont le système, les modules, les fichiers, les classes et les routines. Les informations concernant ces différents aspects sont regroupées dans la classe *Metrics*. *MetricsList* est une classe abstraite où on retrouve les différents objets pour chaque métrique calculée et l'identificateur de la métrique (clé). Chaque sous-classe correspond à toutes les métriques à calculer pour chaque aspect du système; le système, la classe, le fichier, l'interface et la routine.

La classe *DefinedMetric* représente les métriques définies par l'utilisateur. Les classes *SystemMetric*, *ModuleMetric*, *FileMetric*, *ClassMetric, InterfaceMetric* et *RoutineMetric* sont des sous-classes abstraites de la classe *Metric.* Elles servent à leurs tours de classes de base pour les classes qui définissent les métriques. Chaque classe définissant une métrique porte le même nom que la métrique et inclut entre autres une méthode permettant le calcul de la métrique (calculMetric()) et un attribut pour contenir la valeur de la métrique. Une métrique est définie par son identificateur (son nom qui est unique dans le système), sa catégorie (système, classe, interface, module, fichier et routine), sa valeur calculée et le choix de visualisation ou non de la métrique. Les valeurs maximale et minimale que la métrique peut avoir font aussi partie de la définition de la métrique, représentées respectivement par la borne supérieure et la borne

inférieure. Dans les sections qui suivent, nous procédons à la description de ces métriques par aspect.

4.1.1.1.1 Métriques de système

Les métriques de système sont calculées pour tout le système. Douze métriques implantées dans BOAP sont résumées dans le tableau suivant :

Métriques	Descriptions	Observations
AID	Average Inheritance Depth	Correspond au rapport de la somme du DIT (Depth of Inheritance) de toutes les classes et le nombre total de classes.
AWI	Average Width of Inheritance	Correspond à la moyenne de sous-classes d'une classe.
CLS	Number of Classes in a System	Calcule le nombre total de classes du système.
NBTF	Number of Files	Calcule le nombre de fichiers d'un système
NBM	Number of Modules	Représente le nombre de modules constituant le système.
NIC	Number of Independent Classes	Représente le nombre de classes indépendantes. Autrement dit, les classes n'ayant ni parents ni enfants.
TBC	Total Base Classes of System	Compte le nombre de classes n'ayant pas de parents, i.e., qui se trouvent à la racine.
AWII	Average Width of Inheritance for an Interface	C'est le rapport de la somme des sous interfaces par rapport au nombre d'interfaces.
NIS	Number of Interfaces in a System	Compte le nombre total d'interfaces dans le système.
NII	Number of Independent Interfaces	Somme le nombre d'interfaces ne possédant aucun parent et aucun enfant.
TBI	Total Base Interfaces of system	Somme le nombre d'interfaces ne possédant aucun parent.

Tableau 1 : Liste des métriques de système de BOAP

4.1.1.1.2 Métriques de modules

Tel que cité avant, un module représente un package en Java. Ces métriques sont calculées pour chaque paquetage constituant le système. Une seule métrique est implantée ; il s'agit de la métrique NBF (Number of Files). Elle compte le nombre de fichiers dans un package.

4.1.1.1.3 Métriques de classes

Elles correspondent aux métriques de caractéristiques de classes dont les méthodes, les attributs, les descendants, etc. Elles sont au nombre de seize décrites dans le tableau suivant :

Métriques	Descriptions	Observations
CIS	Class Interface Size	Calcule le nombre de méthodes publiques d'une classe. Cette métrique englobe l'ensemble des méthodes nouvellement définies et les méthodes héritées de la classe.
CSM	Class Size Metric	Calcule le nombre de méthodes et d'attributs privés, protégés et publics dans une classe. Cette métrique représente la somme des métriques NOD et NOM.
DIT	Depth of Inheritance Tree	Représente la profondeur maximale de la classe dans l'arbre d'héritage. le DIT est calculé en fonction du nombre de super-classes pour atteindre la racine de l'arbre d'héritage. La valeur de cette dernière est zéro.
NOC	Number Of Children	Correspond au nombre de sous-classes (enfants) directement liées à une classe dans l'arbre d'héritage.
NOD	Number Of Attributes in a class	Correspond au nombre d'attributs hérités et nouveaux. C'est la somme de tous les attributs privés, protégés et publics qui sont hérités et les attributs privés, protégés et publics qui ne le sont pas.
NOM	Number Of Methods in a class	Compte le nombre de méthodes d'une classe. Cette métrique est obtenue en sommant les totaux des deux catégories de méthodes : héritées et non héritées. Chaque catégorie est calculée en fonction de ses méthodes privées, protégées et publiques.
ACAIC	Ancestor Class-Attribute Import Coupling	$ACAIC(c) = \{a / a \in Ai(c) \wedge T(a) \in Ancêtres(c)\}$ C'est la cardinalité de l'ensemble des attributs implémentés dans la classe c : Ai(c). Les types de ces attributs font partie des ancêtres de la classe c.
OCAIC	Others Class-Attribute Import Coupling	Compte le nombre d'attributs qu'une classe particulière possède de type une des classes du système. Ces dernières n'existent ni parmi les descendants ni parmi les ancêtres de cette classe.
DCAEC	Descendents Class-Attribute Export Coupling	Représente le nombre de descendants d'une classe particulière et qui possèdent un attribut de type cette classe.
OCAEC	Others Class-Attribute Export Coupling	Représente le nombre de classes autres que les descendants et les ancêtres de la classe observée. Ces classes possèdent un attribut de type la classe observée.
ACMIC	Ancestors Class-Method Import Coupling	C'est le nombre de paramètres de nouvelles méthodes que la classe observée possède de type une des classes du système. Ces dernières doivent être parmi l'ensemble des ancêtres de la classe observée.
OCMIC	Others Class-Method Import Coupling	C'est le nombre de méthodes n'appartenant pas à l'ensemble des ancêtres et descendants de la classe observée ayant comme paramètre cette dernière.
DCMEC	Descendants Class-Method Export Coupling	Correspond au nombre de méthodes existantes parmi les descendants d'une classe particulière et qui possèdent un paramètre de type cette classe.
OCMEC	Others Class-Method Export Coupling	Correspond au nombre de classes n'appartenant ni aux ancêtres ni aux descendants d'une classe particulière, et qui possèdent une méthode avec un paramètre de type cette classe.

Tableau 2 : Liste des métriques de classes de BOAP

4.1.1.1.4 Métriques d'interfaces

Une seule métrique d'interface est implantée ; il s'agit de la métrique DITI (Depth of Inheritance Tree of Interface). Elle mesure la profondeur maximale de l'interface dans l'arbre d'héritage.

4.1.1.1.5 Métriques de routines

Les métriques de routines touchent les corps des méthodes. En particulier, les différents types d'instructions d'une méthode. Elles sont au nombre de seize et se présentent comme suit :

Métriques	Descriptions	Observations
NOA	Number Of Arguments	Compte le nombre d'arguments d'une méthode.
NBIF	Number Of IF statements	Compte le nombre d'arguments de type IF d'une méthode. Cette métrique distingue entre le nombre d'instruction IF n'ayant pas une instruction ELSE, DANGLING IF et le nombre d'instructions IF qui ont une instruction ELSE.
NBRT	Number Of RETURN statements	Compte le nombre d'arguments de type RETURN dans une méthode.
NBCT	Number Of CONTINUE statements	Compte le nombre d'arguments de type CONTINUE d'une méthode.
NBBK	Number Of BREAK statements	Somme le nombre d'arguments de type BREAK d'une méthode.
NBSW	Number Of SWITCH statements	Somme le nombre d'arguments de type SWITCH d'une méthode.
NBFOR	Number Of FOR statements	Somme le nombre d'arguments de type FOR d'une méthode.
NBWL	Number Of WHILE statements	Calcule le nombre d'arguments de type WHILE d'une méthode.
NBDWL	Number Of DO WHILE statements	Calcule le nombre d'arguments de type DO-WHILE d'une méthode.
NBTH	Number Of THROW statements	Calcule le nombre d'arguments de type THROW d'une méthode.
NCFS	Number Of Control-Flow Statements	Correspond au nombre d'arguments de contrôle dans une méthode. NCFS inclut l'ensemble des instructions IF, RETURN, BREAK, FOR, WHILE, etc. dans une méthode.
STMT	Number of Statements	Correspond au nombre d'instructions dans une méthode. Cette valeur est obtenue en comptant toutes les instructions.
NELO	Number Of maximum nested Loop	Correspond à la profondeur maximale de boucles imbriquées parmi toutes les boucles à l'intérieur de la méthode.
NECO	Number Of maximum nested Condition	Représente la profondeur maximale d'instruction de condition à l'intérieur de la méthode. On compte la profondeur maximale tous types de conditions confondus.
NBNE	Number Of maximum nested condition	Représente la profondeur maximale d'imbrication à l'intérieur d'une méthode. On compte la profondeur maximale pour tous types de boucles et d'instructions de conditions.
NIIL	Number of Instantiation In Loop	Représente le nombre d'instanciations de variables à l'intérieur de la boucle.

Tableau 3 : Liste des métriques de routines de BOAP

4.1.1.2 Analyse et discussion

Dans le chapitre de l'état de l'art, section 2.3, nous avons identifié certains travaux de recherche qui ont un grand apport dans le domaine des métriques logicielles. Ces métriques [2, 3, 16, 20, 22, 26, 49], regroupées dans un tableau en annexe (Annexe 1), confrontées à celles du module de métriques de BOAP font l'objet de l'analyse et de la discussion menées dans cette section pour faire ressortir les métriques à implémenter.

Les métriques du système de BOAP donnent une vue générale sur la nature du système, sa taille ainsi que sa complexité. À titre d'exemple, le nombre de modules (NBM), le nombre de fichiers (NBTF), le nombre de classes (CLS) et le nombre de classes indépendantes (TBC) dans le système. D'autres permettent d'avoir une idée sur le degré d'implantation du concept de l'héritage ; il s'agit des métriques AID et AWI. Par ailleurs, un ensemble de métriques relatives au nombre de méthodes et d'attributs dans une classe sont implantées dans BOAP, ainsi que les métriques permettant de mesurer le couplage entre classe-attribut et classe-méthode [16]. Les métriques de routines implantées dans BOAP se concentrent sur la méthode. Elles permettent d'avoir une idée sur la complexité du traitement d'une méthode. Par exemple, le nombre d'instructions exprimé par la métrique STMT permet d'avoir une idée sur la taille d'une méthode.
De façon générale, certaines métriques de l'état de l'art ont été directement implantées dans le module de métriques de BOAP ; il s'agit des métriques NOC et DIT de Chidamber et Kemerer [22], et des métriques qui capturent le couplage de type classe-attribut (ACAIC, OCAIC, DCAEC, OCAEC) et classe-méthode (ACMIC, OCMIC, DCMEC, OCMEC) de Briand *et al.* [16]. D'autres métriques existent dont les définitions sont inspirées de celles existantes dans la littérature avec des modifications légères telles que la métrique NOM (Number Of Methods in a class) de BOAP. Elle est définie par le nombre de méthodes héritées et nouvelles d'une classe, alors que dans la littérature NOM est définie par le nombre de méthodes locales [49].

Cependant, certaines métriques de couplage ne sont pas implantées dans BOAP telles que le couplage entre objets exprimé par la métrique CBO, le couplage sans tenir compte de la relation d'héritage exprimé par la métrique CBO' et le nombre de méthodes invoquées en réponse à un message exprimé par la métrique RFC [21, 22]. Plus encore, les métriques qui expriment l'interaction de type méthode-méthode (AMMIC, OMMIC, DMMEC, OMMEC) de Briand *et al.* [16], les métriques de couplage conçues par Kabaili [43] et qui sont dérivées de celles de

Chidamber et Kemerer (NOC*, CBO_IUB et CBO_U) et les métriques de couplage (MPC et DAC) de Li et Henry [49] ne sont pas aussi implantées dans le module de métriques de BOAP. De même que les métriques qui capturent différents aspects de l'approche orientée objet dont l'héritage, l'encapsulation et le polymorphisme (AHF, AIF, MHF, MIF, PF) de Brito e Abreu et Carapuça [3].

4.1.1.3 Conclusion

En général, la version actuelle du module de métriques de BOAP ne traite pas les métriques de couplage. En particulier, celles qui permettent de calculer le couplage à travers les méthodes et les métriques qui réfèrent aux mécanismes de base du paradigme orienté objet comme l'encapsulation, l'héritage et le polymorphisme. Notre travail consiste donc à compléter le module des métriques de BOAP en implémentant ces nouvelles métriques. Le tableau suivant liste toutes les métriques à implémenter.

Métriques	Descriptions
NAS	Number of Attributes in the System
NMS	Number of Methods in the System
AIF	Attribute Inheritance Factor
AHF	Attribute Hiding Factor
MIF	Method Inheritance Factor
MHF	Method Hiding Factor
PF	Polymorphism Factor
NOC*	Number Of Children in sub-tree
DAC	Data Abstraction Coupling
DAC'	Data Abstraction Coupling
WMC	Weighted Methods per Class
CBO	Coupling Between Object
RFC	Response For a Class
MPC	Message Passing Coupling
AMMIC	Ancestors Method -Method Import Coupling
OMMIC	Others Method –Method Import Coupling
DMMEC	Descendants Method - Method Export Coupling
OMMEC	Others Method – Method Export Coupling
CBO'	Coupling Between Object
CBO-IUB	CBO Is Used By
CBO-U	CBO Using

Tableau 4 : Liste des métriques à implémenter

Cependant, face au problème de modèle présent dans BOAP et limitant l'implémentation des métriques relatives au couplage, nous nous somme limités à l'implémentation des métriques présentées dans la section suivante.

4.1.2 Implémentation de l'extraction des métriques

Cette section présente les changements importants apportés au module de métriques de BOAP. C'est une extension de la version existante pour pouvoir supporter plus de métriques. Cette extension a bien sûr engendré des ajouts et des modifications. Dans la suite de cette section, nous présentons ces changements et les descriptions des métriques ajoutées.

4.1.2.1 Métriques de système

L'ajout de nouvelles métriques au module de métriques de BOAP a pour effet l'ajout de nouvelles classes et des changements dans d'autres. Afin d'intégrer ces métriques dans le module, nous avons procédé par :

1. L'ajout de sous-classes de la classe *SystemMetric.* Ces classes portent les noms des métriques qui sont implémentées et se présentent comme suit :
   - NAS : pour calculer la métrique "Number of Attribute in the System" ;
   - NMS: pour calculer la métrique "Number of Method in the System" ;
   - AIF : pour calculer la métrique "Attribute Inheritance Factor" ;
   - MIF : pour calculer la métrique "Method Inheritance Factor" ;
   - AHF : pour calculer la métrique "Attribute Hiding Factor " ;
   - MHF : pour calculer la métrique "Method Hiding Factor " ;
   - PF : pour calculer la métrique " Polymorphisme Factor ".

Pour chacune des classes ajoutées ci haut, une méthode est définie permettant de calculer la valeur de cette métrique ; il s'agit de la méthode *calculMetric().*

2. La mise à jour de la classe *SystemMetricList* en deux étapes :
   - Ajout des identificateurs des métriques en lettres majuscules dans metricsIdentifers[]; il s'agit de NAS, NMS, AIF, MIF, AHF, MHF et PF.
   - Ajout d'une méthode ayant comme nom : calcul + nom méthode (lettres en majuscules) pour chaque identificateur ajouté. Exemple : calculNAS().

Description des métriques implantées

NAS (Number of Attributes per System) :

Cette classe permet de calculer le nombre d'attributs définis dans un système. Le calcul de cette métrique se fait par la sommation du nombre d'attributs de chaque classe du système. Ainsi, si nous représentons par C le nombre de classes du système S et par $A(C_i)$ le nombre d'attributs

de la classe C_i, le nombre d'attributs du système est:

$$NAS(S) = \sum_{i=1}^{C} A(C_i)$$

NMS (Number of Methods per System) :

Cette classe permet de calculer le nombre de méthodes définies dans le système. Ce nombre est calculé par la sommation des méthodes de chaque classe du système. Ainsi, si nous représentons le système par S et par $M(C_i)$ le nombre de méthodes de la classe C_i, alors le nombre de méthodes du système est calculé par :

$$NMS(S) = \sum_{i=1}^{C} M(C_i)$$

MHF (Method Hiding Factor) :

Représente le pourcentage de méthodes cachées [3]. C'est le ratio de la somme des invisibilités de toutes les méthodes définies dans toutes les classes du système et du nombre total de méthodes définies dans le système. L'invisibilité d'une méthode est le pourcentage du total des classes à partir desquelles cette méthode n'est pas visible. La partie de la classe visible pour les autres classes du système est représentée par les méthodes de la classe déclarées publiques et la partie de la classe invisible (hidden) est représentée par les méthodes déclarées privées et-ou protégées. Cette métrique est exprimée par :

$$MHF(S) = \frac{\sum_{i=1}^{C} M_h(C_i)}{\sum_{i=1}^{C} M_d(C_i)}$$

Avec pour une classe C_i du système S :

- $M_v(C_i)$ est le nombre de méthodes visibles de la classe C_i ;
- $M_h(C_i)$ est le nombre de méthodes invisibles (hidden) de la classe C_i ;
- $M_d(C_i)$ est le nombre de méthodes définies dans la classe C_i donné par :

$$M_d(C_i) = M_v(C_i) + M_h(C_i)$$

AHF (Attribute Hiding Factor) :

Représente le pourcentage d'attributs cachés [3]. Elle est définie comme le ratio de la somme des invisibilités des attributs définis dans toutes les classes du système et du nombre total des attributs définis dans le système. Les définitions utilisées pour MHF sont adoptées pour les attributs (Au lieu de M pour méthodes, nous remplaçons les formules avec A pour les adaptées aux attributs).

$$AHF(S) = \frac{\sum_{i=1}^{C} A_h(C_i)}{\sum_{i=1}^{C} A_d(C_i)}$$

MIF (Method Inheritance Factor) :

Cette classe représente le pourcentage de méthodes héritées [3]. Cette métrique est définie comme le ratio de la somme des méthodes héritées de toutes les classes du système et du nombre total de méthodes de toutes les classes (localement définies plus celles héritées) du système. Cette métrique est exprimée par :

$$MIF(S) = \frac{\sum_{i=1}^{C} M_i(C_i)}{\sum_{i=1}^{C} M_a(C_i)}$$

Où pour une classe du système S :

- $M_n(C_i)$ est le nombre de nouvelles méthodes définies dans C_i (méthodes qui ne redéfinissent pas les méthodes héritées) ;
- $M_i(C_i)$ est le nombre de méthodes héritées dans C_i (non redéfinies) ;
- $M_o(C_i)$ est le nombre de méthodes redéfinies dans C_i (héritées qui sont redéfinies) ;
- $M_a(C_i)$ est le nombre de méthodes disponibles dans C_i (celles définies dans la classe C_i et celles héritées) : $M_a(C_i) = M_n(C_i) + M_i(C_i) + M_o(C_i)$

AIF (Attribute Inheritance Factor) :

Cette classe permet de calculer le pourcentage d'attributs hérités [3]. C'est le ratio de la somme des attributs hérités dans toutes les classes du système et du nombre total des attributs de toutes les classes (localement définis plus ceux hérités) du système. Les définitions utilisées pour MIF sont adoptées pour les attributs (Au lieu de M pour méthodes, nous remplaçons les formules avec A pour les adaptées aux attributs). Ainsi, pour le système S, AIF est représenté par :

$$AIF(S) = \frac{\sum_{i=1}^{C} A_i(C_i)}{\sum_{i=1}^{C} A_a(C_i)}$$

PF (Polymorphism Factor) :

Cette métrique calcule le pourcentage de méthodes polymorphes par rapport au nombre total de méthodes potentiellement polymorphes [3]. Elle représente le ratio du nombre actuel des différentes situations possibles de polymorphisme pour chaque classe du système et du nombre

de situations distinctes de polymorphisme de chaque classe du système. Cette métrique est exprimée par :

$$PF(S) = \frac{\sum_{i=1}^{C} M_o(C_i)}{\sum_{i=1}^{C} [M_n(C_i) * DC(C_i)]}$$

Où les mêmes définitions établies pour MIF sont utilisées et $DC(C_i)$ représente le nombre de descendants de la classe C_i.

4.1.2.2 Métriques de classes

Comme pour les métriques de système, l'intégration des métriques de classes dans le module de BOAP suit les mêmes étapes :

1. Ajout de sous-classes de la classe *ClassMetric*. Ces classes portent les noms des métriques qui sont implémentées et se présentent comme suit :
   - DAC : pour calculer la métrique "Data Abstract Coupling" ;
   - DAC' : pour calculer la métrique " Data Abstract Coupling " ;
   - NOC*: pour calculer la métrique "Number of Children in sub-tree " ;
   - WMC : pour calculer la métrique "Weighted Method per Class ".

Pour chaque classe du système, une méthode *calculMetric()* est définie permettant de calculer la valeur de la métrique.

2. Changement de la classe *ClassMetricList* par ajout :
   - D'identificateurs de métriques en lettres majuscules dans l'attribut metricsIdentifers[]; il s'agit de DAC, DAC', NOC* et WMC.
   - D'une méthode ayant comme nom : calcul + nom méthode (lettres en majuscules) pour chaque métrique ajoutée. Exemple : calculWMC().

Description des métriques implantées

NOC* (Number Of Children in sub-tree) :

Dérivée de la métrique NOC de [22]. Cette classe permet de calculer le nombre d'enfants dans le sous-arbre d'héritage [43]. Considérons une classe C_i du système et $DC(C_i)$ le nombre de descendants de la classe C_i, alors $NOC^*(C_i) = DC(C_i)$.

DAC (Data Abstraction Coupling) :

Elle correspond au nombre d'ADT (Abstract Data Type) définis dans une classe [49]. Cette métrique représente le nombre d'attributs non hérités qui sont de type une des classes du système. Considérons C l'ensemble des classes constituant le système S, $AI(C_i)$ l'ensemble des attributs implémentés (les attributs non hérités) d'une classe $C_i \in$ C et par T (a) le type d'attribut. Alors $DAC(C_i)$ est représenté par :

$$DAC\ (C_i) = \{a \mid a \in AI\ (C_i) et T\ (a) \in C\}$$

DAC' (Data Abstraction Coupling) :

Même principe que DAC. Cependant, au lieu de compter le nombre d'attributs qui sont de type une des classes du système, nous comptons le nombre de classes utilisées comme types d'attributs définis dans la classe C_i [18]. Elle est représentée par :

$$DAC'(C_i) = \{T(a) \mid a \in AI\ (C_i) et T\ (a) \in C\}$$

WMC (Weighted Methods per Class) :

Elle représente la somme des complexités de toutes les méthodes d'une classe [22]. Nous utilisons la valeur 1 comme complexité de la méthode. Pour une classe C_i, WMC représente l'ensemble des méthodes locales définies dans cette classe.

4.1.3 Conclusion

Le travail élaboré dans cette section a permis d'étendre l'ancienne version du module de métriques de BOAP pour supporter d'autres métriques. Il est réparti en deux grandes sections. Dans la première, nous avons fait un tour d'horizon des métriques offertes par le système BOAP. Dans la description du module de métriques de BOAP, nous nous somme basés sur [31, 32]. Ensuite, nous avons mené une analyse et discussion dans le but de faire ressortir les métriques à implémenter. Dans la deuxième section, nous avons décrit les métriques ajoutées. Ces dernières ont été intégrées dans le module de métriques de BOAP et le test du module en entier est réalisé avec plusieurs systèmes.

Actuellement, le problème de modèle de BOAP limite l'implémentation des métriques qui capturent le couplage entre méthodes. Afin d'atteindre les objectifs de ce mémoire et de pallier à ce problème, nous avons développé un outil sous la plate-forme Eclipse qui permet d'analyser un code source Java et d'extraire des métriques.

4.2 Extraction des métriques avec ECLIPSE

"Eclipse is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools" [30]. C'est une plate-forme modulaire pour le développement d'environnements de développement. La principale caractéristique d'Eclipse est l'extensibilité de l'environnement. Il est construit à base de mécanismes capables de charger, d'intégrer et d'exécuter des modules appelés extensions (plug-ins). Une extension est une unité de fonction qui peut être développée et délivrée séparément. À titre d'exemple, l'outil de développement Java JDT (Java Development Tools) et l'environnement de développement de plug-in PDE (Plug-in Development Environment) sont des extensions. Cette dernière intéresse principalement les développeurs qui veulent étendre Eclipse, puisqu'il permet de développer et d'intégrer de nouveaux outils à son environnement.

Cette plate-forme nous a servi de base pour le développement et à l'intégration du nouvel outil qui permet d'analyser un code source Java et d'extraire des métriques. Dans les sections suivantes, on présente le processus d'analyse du code source et le travail réalisé sous cette plate-forme.

4.2.1 Processus d'analyse du code source

Eclipse offre une extension qui offre le moyen d'analyser (parser) un fichier source Java et de fournir son arbre syntaxique. En se basant sur ce principe, on a construit notre outil qui permet d'analyser le code source des systèmes orientés objets écrites en langage Java et d'extraire des métriques de couplage. Pour atteindre cet objectif, le processus suivi est schématisé dans la figure suivante :

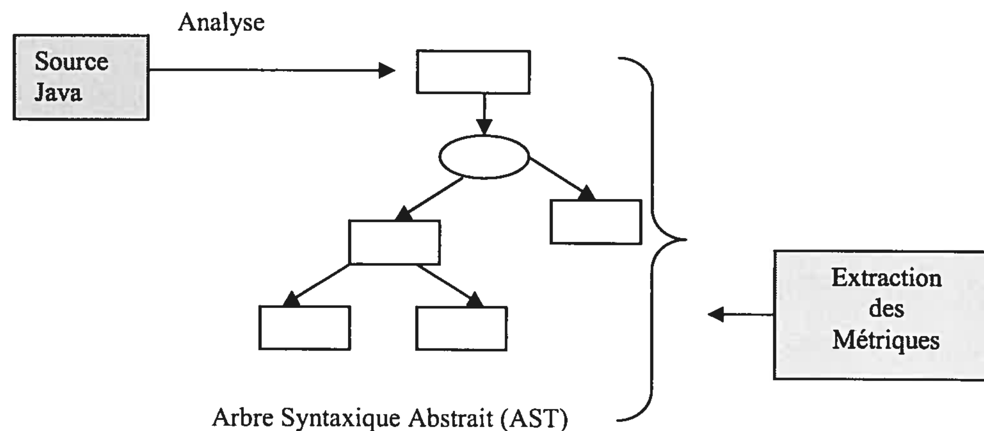

Figure 8 : Processus d'analyse et d'extraction de métriques

L'arbre syntaxique (AST) correspond à un ensemble de nœuds représentant tous les constituants de la classe jusqu'aux instructions ; c'est une structure arborescente d'objets. Ainsi, pour une unité de compilation (fichier source Java) constitué d'un ensemble de classes, l'arbre se présente comme suit :

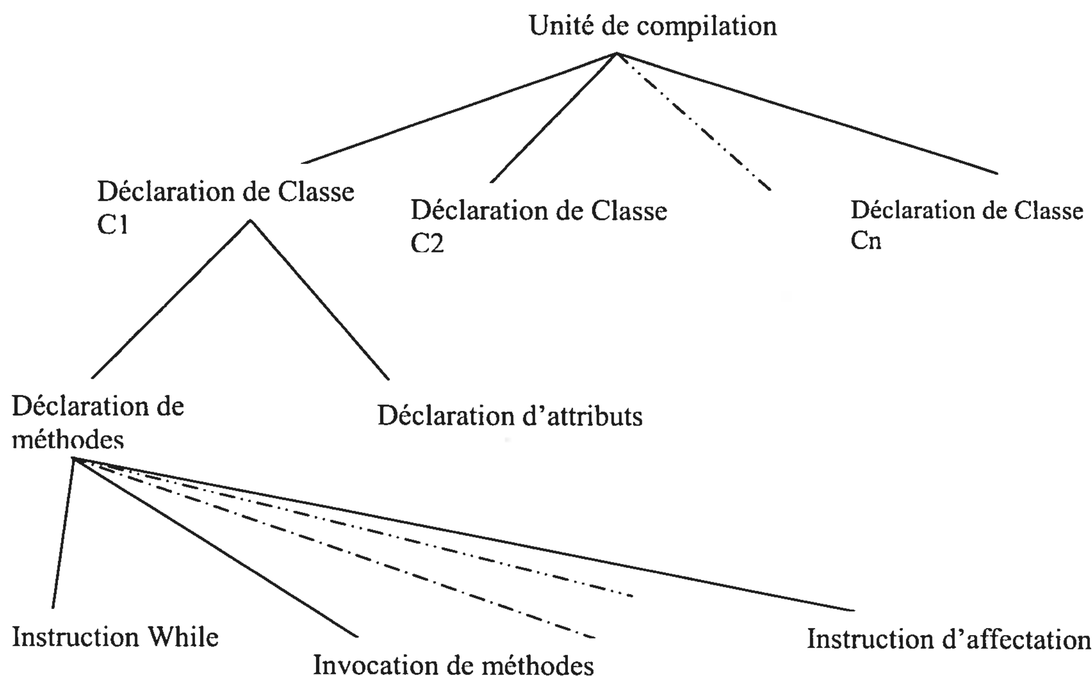

Figure 9 : Arbre Syntaxique Abstrait (AST)

Dans notre travail, nous procédons par parcourir l'arborescence du projet, fichier par fichier, l'analyser et faire ressortir les informations utiles pour l'implémentation des métriques, à savoir : les classes, les super-classes, les descendants, les attributs, les méthodes, les appels de fonctions, etc. Pour cela, nous avons utilisé la notion de patron de conception Visiteur (Visitor).

Le visiteur représente *"an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."* [33]. Un visiteur est une classe qui visite d'autres objets et effectue des tâches appropriées. Le visiteur de l'arbre syntaxique d'Eclipse : "*ASTVisitor*" est utilisé pour examiner la structure d'une unité de compilation, i.e., son arbre syntaxique jusqu'au niveau instruction.

ASTVisitor est une classe abstraite à laquelle est associée une paire de méthodes "visit(T node)" et "endVisit(T node)" pour les différentes catégories de nœuds de l'arbre syntaxique ; il s'agit de

l'accès à un tableau, l'affectation, les blocs d'instructions, les instructions break et continue, les boucles for, while, etc.

- Visit(T node) : permet de visiter le nœud (node) de l'AST et d'effectuer certaines opérations. Si la valeur retournée est vraie (True), alors les nœuds enfants seront visités. Cette classe ne fournit aucune implémentation, des sous-classes doivent être définies et implémentent la méthode selon le besoin. Par exemple : visit(CompilationUnit node) visite le fichier source Java, visit(MethodInvocation node) visite les appels de fonctions et visit(IfStatement node) visite les instructions if.
- EndVisit (T node) : cette méthode est appelée après que tous les nœuds enfants aient été visités. L'implémentation par défaut de cette classe ne fait aucune tâche. Des sous-classes doivent être définies pour réimplanter la méthode.

4.2.2 Travail effectué sous Eclipse

Au fil de cette section, on présente la terminologie et le formalisme ainsi que la description des métriques implémentées sous Eclipse.

4.2.2.1 Terminologie et formalisme

Depuis un certain temps, les recherches dans le domaine des mesures du logiciel ont souffert du manque de terminologie standard et de formalisme pour définir les mesures de manière non ambiguë et complètement opérationnelle [18]. Par conséquent, le développement de prédicateurs de qualité de logiciel consistants, compréhensibles et significatifs a été sévèrement entravé. D'où la nécessité d'un consensus sur la terminologie et d'un formalise à définir et à suivre dans l'expression des mesures logicielles. De notre part, pour décrire les métriques de conception à implémenter, nous avons utilisé la terminologie et le formalise décrit ci-dessous en s'inspirant de celui de Briand *et al.* [18].

Système :

Un système orienté objet est composé d'un ensemble de classes noté par C. Pour représenter les informations relatives à la relation d'héritage qui peut exister entre les classes du système, nous avons établi ces définitions. Ainsi, pour chaque classe $c \in C$ on a :

- Parents(c) $\subset$ C : représente l'ensemble des parents de la classe c ;
- Childrens(c) $\subset$ C : représente l'ensemble des enfants de la classe c ;
- Ancestors (c) $\subset$ C : représente l'ensemble des ancêtres de la classe c ;

- Descendants(c) $\subset$ C : représente l'ensemble des descendants de la classe c ;
- Others(c) $\subset$ C : représente l'ensemble des classes qui ne sont ni ancêtres ni descendants de la classe c.

Méthodes :

Une classe c est composée d'un ensemble de méthodes noté par M(c), avec $M(c)=Md(c) \cup Mi(c)$ et $Md(c) \cap Mi(c) = \phi$.

- Mi(c) : représente les méthodes implémentées dans la classe c, i.e., les méthodes héritées et qui sont redéfinies;
- Md(c) : représente les méthodes déclarées dans la classe c, i.e., les méthodes héritées mais qui ne sont pas redéfinies.

Une méthode peut être héritée, redéfinie ou nouvelle, avec $M(c)=Minh(c) \cup Movr(c) \cup Mnew(c)$. Nous représentons respectivement ces catégories par :

- Minh(c) : ensemble des méthodes héritées de la classe c ;
- Movr(c) : ensemble des méthodes redéfinies de la classe c ;
- Mnew(c) : ensemble de nouvelles méthodes de la classe c (méthodes non redéfinies et non héritées de la classe c).

Chaque méthode possède une liste de paramètres représentée par Par(m). L'ensemble des méthodes du système est représenté par M(C).

Attributs :

Une classe c est composée d'attributs qui peuvent être hérités ou nouveaux représenté par $A(c) = Ad(c) \cup Ai(c)$.

- Ai(c) : ensemble des attributs implémentés dans la classe c (les attributs non hérités).
- Ad(c) : ensemble des attributs déclarés dans la classe c (les attributs hérités).

Chaque attribut possède un type représenté par T (a).

Invocation de méthodes :

Pour définir le couplage d'une classe, il est nécessaire de définir l'ensemble de méthodes invoquées par une méthode $m \in M(c)$ et les fréquences de ces invocations.

- SIM (m), Statically Invoked Methods of m : Considérant $c \in C$, $m \in Mi(c)$ et $m' \in M(C)$. Alors $m' \in SIM(m) \Leftrightarrow \exists d \in C$ tel que $m' \in M(d)$ et le corps de la méthode m possède une invocation de méthode où m'est invoquée pour un objet de type statique d.
- NSI (m, m'), Number of Static Invocations of m' by m : Considérant $c \in C$, $m \in Mi(c)$ et $m' \in SIM(m)$. NSI (m, m') est le nombre d'invocations de méthodes dans m où m'est invoquée pour un objet de type statique de la classe d et $m' \in M(d)$.

Pour compter le type d'interaction méthode–méthode entre les classes c et d du système, nous utilisons MMI(c, d).

$$MMI(c,d) = \sum_{m \in Mi(c)} \sum_{m' \in Mnew(d) \cup Movr(d)} NSI(m, m')$$

Prédicats :

Pour compter les invocations entre deux classes $c \in C$ et $d \in C$ nous utilisons :

Uses (c, d) <=> $(\exists m \in Mi(c): \exists m' \in Mi(d): m' \in SIM(m)) \cup (\exists m \in Mi(c): \exists a \in Ai(d): \exists a \in AR(m))$

AR (m) représente l'ensemble des attributs référencés par la méthode $m \in M(c)$.

4.2.2.2 Description des métriques implantées

La définition des métriques se base sur les articles de Chidamber et Kemerer dans [21, 22], de Briand *et al.* [16, 20], Li et Henry [49] et Hind Kabaili dans sa thèse [43]. Dans le reste du chapitre, nous présentons les définitions des différentes métriques implémentées.

CBO (Coupling Between Object) :

$$CBO(c) = |\{d \in C - \{c\} \mid uses(c,d) \cup uses(d,c)\}|$$

Représente le nombre de classes avec lesquelles une classe est couplée. Deux classes sont dites couplées, si les méthodes définies dans l'une utilisent des méthodes ou des variables d'instances de l'autre [22]. Une classe c est couplée à une classe d si elle utilise d ou est utilisée par d. La définition du prédicat uses (c,d) demande que la méthode qui utilise la classe d et la méthode utilisée ou l'attribut de la classe d soient implémentés dans leurs classes [18].

CBO' (Coupling Between Object) :

$$CBO'(c) = |\{d \in C - (\{c\} \cup Ancestors(C)) \mid uses(c,d) \cup uses(d,c)\}|$$

Définie dans [21], compte le nombre de classes avec lesquelles une classe c est couplée et n'ayant pas une relation d'héritage. Cette définition ne prend pas en considération l'ensemble des ancêtres d'une classe.

CBO-IUB (CBO-Is Used By) :

$$CBO_IUB(c) = |\{d \in C - \{c\} \mid uses(d,c)\}|$$

Dérivée de la métrique CBO [22]. La métrique CBO compte les interactions dans les deux sens : « utilise » et « utilisée par ». Cette métrique consiste au nombre de classes utilisant la classe cible c, i.e., les classes qui utilisent une méthode ou une variable d'instance de la classe c [43].

CBO-U (CBO Using) :

$$CBO_U(c) = |\{d \in C - \{c\} \mid uses(c,d)\}|$$

Dérivée de la métrique CBO [22]. Représente la partie du CBO qui s'intéresse aux classes utilisées par la classe cible c, i.e., c utilise d, si les méthodes de la classe c utilisent des méthodes ou des variables d'instances de la classe d [43].

RFC (Response For a Class) :

$$RFC(c) = \{M\} \cup \{R_i\}_{all\,i}$$

{M} est l'ensemble des méthodes de la classe.

{Ri} l'ensemble des méthodes appelées par la méthode i.

Représente le nombre de méthodes invoquées en réponse à un message. Établie par Chidamber et Kemerer dans [22], elle vaut le nombre de méthodes de la classe et le nombre de méthodes appelées par chaque méthode m de la classe c.

MPC (Message Passing Coupling) :

$$MPC(c) = \sum_{m \in Mi(c)} \sum_{m' \in SIM(m) - Mi(c)} NSI(m, m')$$

Définie par Li et Henry comme le nombre de messages envoyés par une classe vers d'autres classes [49]. MPC(c) compte seulement les invocations de méthodes des autres classes et non pas les invocations de méthodes propres à la classe c [18].

AMMIC (Ancestors Method -Method Import Coupling) :

$$AMMIC(c) = \sum_{d \in Ancestors(c)} MMI(c, d)$$

Correspond au nombre de classes parentes avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type IC. Pour chaque méthode de la classe c, on cherche si elle appelle une méthode des autres classes. Ces dernières représentent l'ensemble des ancêtres.

OMMIC (Others Method –Method Import Coupling) :

$$OMMIC(c) = \sum_{d \in Others(c)} MMI(c, d)$$

Correspond au nombre de classes (autres que les super-classes et les sous-classes) avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type IC.

DMMEC (Descendants Method - Method Export Coupling) :

$$DMMEC(c) = \sum_{d \in Descendants(c)} MMI(d, c)$$

Correspond au nombre de sous-classes avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type EC.

OMMEC (Others Method - Method Export Coupling) :

$$OMMEC(c) = \sum_{d \in Others(c)} MMI(d, c)$$

Correspond au nombre de classes (autres que les super-classes et les sous-classes) avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type EC.

4.3 Conclusion

Dans ce chapitre, nous avons procédé dans un premier temps à l'extension du module de métriques de BOAP par l'ajout de nouvelles métriques. Face au problème de modèle que présente cette plate-forme et compte tenu de l'objectif de ce mémoire, nous avons développé sous la plate-forme Eclipse un outil permettant d'analyser un code source Java et d'extraire d'autres métriques, en particulier, celles qui permettent de mesurer le couplage. Ces deux outils ont servi dans le cadre de notre projet, spécialement dans le calcul des métriques pour leur validation en ce qui concerne l'analyse d'impact du changement. L'outil sous Eclipse doit être vu comme étant un ajout plutôt qu'une réécriture. La raison est que d'une part, ce nouveau outil utilise une plate-forme de plus en plus répandue, et que d'autre part, il peut être intégré dans un autre système comme le dicte le principe d'extension en Eclipse.

Nous avons mené de nombreux tests sur des codes sources de systèmes de tailles différentes, et ce pour vérifier la complétude et la validité des deux outils d'extraction de métriques. Ces tests sont faits conjointement avec des calculs effectués par les développeurs. Suite aux résultats positifs obtenus, nous avons décidé d'utiliser ces deux outils pour extraire les métriques nécessaires pour l'expérience du chapitre 5 et pour valider empiriquement leur pertinence pour l'analyse d'impact du changement. Dans le chapitre suivant, nous présentons le détail de l'expérience menée ainsi que les résultats obtenus.

Chapitre 5 Expérience et Résultats

Rappelons que notre but est d'estimer l'impact du changement des systèmes orientés objets en maintenance et faire ressortir les mesures de conception qui sont de bons indicateurs d'impact du changement.

Dans le chapitre 3, nous avons exposé notre démarche d'analyse d'impact ; nous avons choisi le modèle statique qualitatif d'impact du changement (section 3.2.2) et formulé des hypothèses liant les propriétés de couplage et d'héritage avec l'impact du changement (section 3.3). Dans le chapitre 4, nous avons présenté les outils d'extraction des métriques ainsi que toutes les métriques qui peuvent être extraites. Dans ce chapitre, nous allons étudier les relations suggérées par ces hypothèses et faire ressortir les métriques de couplage et d'héritage qui sont de bons indicateurs d'impact du changement. Pour cela, nous avons mené une expérience avec la collaboration des développeurs du CRIM. Dans la première section, nous présentons l'environnement de l'expérience, suivi des résultats de calcul des métriques et des impacts des changements. Ensuite, nous exposons une partie sur la construction des modèles d'évaluation de l'impact. Finalement, nous procédons à l'analyse des modèles construits et à l'interprétation des résultats.

5.1 Données de l'étude

Dans cette section, nous présentons les données nécessaires pour réaliser notre expérience ; il s'agit des systèmes de l'étude, de l'ensemble des métriques choisies, de la liste des changements sélectionnés et du questionnaire que nous avons établi afin d'estimer l'impact du changement.

5.1.1 Systèmes étudiés

Dans notre étude, nous utilisons deux systèmes industriels de tailles différentes, développés avec le langage Java, fournis par le CRIM : BOAP et PRECI. Les caractéristiques des deux systèmes sont présentées dans le tableau suivant :

Caractéristiques	Système BOAP	Système PRECI
Nombre de fichiers	424	173
Nombre de modules	22	29
Nombre de classes indépendantes	103	71
Nombre de classes	430	160
Nombre de classes de bases	117	75
Nombre de méthodes	3546	2289
Nombre d'attributs	2247	1977

Tableau 5 : Caractéristiques des systèmes étudiés

5.1.2 Liste des métriques

Dans le chapitre 4 sur l'extraction des métriques, nous avons présenté toutes les métriques de conception orientée objet offertes par les outils d'extraction (BOAP et l'outil développé sous Eclipse). Initialement, plusieurs métriques ont été extraites. Ci-dessous nous listons les métriques sélectionnées afin d'évaluer les hypothèses établies dans la section 3.3 de notre approche d'analyse d'impact.

Hypothèse 1 : la position de la classe dans l'arbre d'héritage d'un système orienté objet affecte l'impact du changement. Les métriques sélectionnées sont comme suit :

- DIT(c), depth of inheritance tree.
- NOC(c), number of children.
- NOC*(c), number of children in sub-tree.

Hypothèse2 : le degré d'interdépendance entre les classes affecte l'impact du changement. Les métriques choisies sont :

- CBO(c), coupling between object classes.
- CBO'(c), coupling between object classes.
- CBOIU(c), coupling is used by.
- CBOU(c), coupling using.
- RFC(c), response for class,
- MPC(c), message-passing coupling.
- DAC(c), data abstraction coupling,
- DAC'(c), data abstraction coupling,

- ACAIC(c), ancestors class-attribute import coupling.
- DCAEC(c), descendants class-attribute export coupling.
- ACMIC(c), ancestors class-method import coupling.
- OCMIC(c), others class-method import coupling.
- DCMEC(c), descendants class-method export coupling.
- OCAEC(c), others class-attribute export coupling.
- OCAIC(c), others class-attribute import coupling.
- OCMEC(c), others class-method export coupling.
- AMMIC(c), ancestors method-method import coupling.
- OMMIC(c), others method-method import coupling.
- DMMEC(c), descendants method-method export coupling.
- OMMEC(c), others method- method export coupling.

5.1.3 Liste des changements

D'après la classification des changements faite dans la section 3.1, nous avons choisi d'étudier des changements parmi ceux ayant les occurrences souvent (section 3.1.2.3) et-ou rare (section 3.1.2.2). Au fil de cette section, nous présentons ces changements avec description de façon intuitive des effets qu'ils peuvent avoir :

1. Changement de l'implémentation d'une méthode : ce changement peut ou ne peut affecter les autres classes du système. Si le changement ne touche pas le comportement et les fonctionnalités de la méthode, alors il n'a aucun effet. Au contraire, si le changement altère les fonctions de la méthode, alors l'effet de ce type de changement affectera la classe changée et peut s'étendre et atteindre toutes les classes (clientes et sous-classes) qui référencent cette méthode.
2. Changement de la portée d'une méthode de publique vers privée : signifie que moins de services sont offerts. En effet, toutes les classes et sous-classes qui utilisent la méthode changée peuvent être affectées par ce changement.
3. Changement de la signature d'une méthode : le changement de la signature d'une méthode peut toucher :
    - le nom de la méthode, ainsi les classes qui référencent cette méthode doivent être examinées et changées.

- la liste des paramètres par ajout ou suppression, alors toutes les classes qui appellent cette méthode doivent être vérifiées pour s'assurer de la validité des appels.

Ainsi, l'effet portera sur toutes les classes (clientes et sous-classes) qui sont en relation avec la classe changée.

4. Suppression d'une classe : quand une classe est supprimée, tous ses attributs et ses méthodes ne sont plus disponibles. Par la suite, toutes les références à cette classe n'auront aucune signification, sauf si ces références sont redirigées vers d'autres classes. Ce type de changement affectera n'importe quelle classe en relation avec la classe changée.
5. Suppression d'une méthode : lorsqu'une méthode est supprimée, cela laisse entendre que ses services ne sont plus valables. Toutes les classes qui la référencent doivent changer en conséquence, que cela soit par changement du sens de l'appel vers une autre méthode existante ou par sa suppression.
6. Changement du type d'un attribut : lorsqu'on change le type d'un attribut, toutes les classes qui référencent cet attribut doivent être examinées afin de s'assurer du bon fonctionnement du code, et qu'il satisfait encore les spécifications initiales.
7. Changement de la portée d'un attribut de publique vers privée : signifie que moins de services sont offerts. En conséquence, des changements dans toutes les classes qui utilisent la classe changée s'imposent.

5.1.4 Questionnaire du calcul d'impact

Nous avons défini l'impact comme le degré de difficulté ressentie pour effectuer le changement (section 3.2.1.1). Pour évaluer les impacts des changements choisis dans la section précédente en utilisant notre modèle statique qualitatif d'impact du changement (section 3.2.2), nous avons établi un questionnaire pour chaque système de l'étude (BOAP et PRECI). Un exemplaire du questionnaire pour le système PRECI est présenté comme suit :

Classes du système PRECI	Types de changements						
	Ch1	Ch2	Ch3	Ch4	Ch5	Ch6	Ch7
ClassdependencePanel.java		IG		IF			
Trace.java							IN
Site.java	IF	IG		IN		IM	
Centrale.java							
Contrainte.java		IN			IN		
....	...	...	......	...	...	...	....

Tableau 6 : Questionnaire du calcul d'impact

Le questionnaire est composé d'une colonne contenant la liste des classes du système de l'étude et d'une autre, composée des différents types de changements. Le remplissage du questionnaire se fait en deux étapes qui sont :

1. Pour chaque classe du système, la personne doit choisir parmi la liste des changements ceux qui s'appliquent à la classe. Une classe peut réunir tous les types de changements, quelques-uns ou aucun type de changement.
2. Pour chaque type de changement identifié dans la première étape, remplir la case correspondante par l'un des types d'impact ressenti suivant : Impact Nul (IN), Impact Faible (IF), Impact Moyen (IM) ou Impact Grand (IG). Un seul type d'impact est valable pour chaque type de changement qui se présente pour la classe.

Une telle démarche est clarifiée par l'exemple suivant : si la classe « Site.java » a subi les changements Ch1, Ch2, Ch4 et Ch6, alors au dessous de chaque type il faut inscrire l'impact ressenti. Dans ce cas, la classe « Site.java » se retrouve avec quatre types d'impact, chacun correspond à un type de changement donné ; il s'agit respectivement de IF, IG, IN et IM.

Nous disposons de deux projets développés par des développeurs informaticiens et qui sont en phase de maintenance perfective. Le questionnaire est fourni à quatre personnes qui assurent la maintenance des deux systèmes au CRIM. Ces personnes ont déjà travaillé sur des projets industriels de grandes tailles et ils sont familiers avec le langage Java. Il est important de souligner que les sujets n'ont jamais effectué une expérience similaire. Avant de donner le questionnaire, nous avons expliqué aux sujets le but de l'expérience, à quel type d'exercice ils ont à faire, les données sur lesquelles ils vont travailler et quel genre de réponses ils ont à fournir.

Chaque personne est responsable de la valeur de l'impact affectée. Cependant, vu le nombre important de classes à analyser, quelques imprécisions peuvent y être introduites. Ainsi, pour éviter les réponses incorrectes et éliminer l'effet de la fatigue, un nombre de jours suffisant est offert aux sujets afin d'accomplir la tâche de façon adéquate.

5.2 Résultats

Nous avons mis l'accent sur les métriques d'héritage et de couplage comme métriques influençant la caractéristique à étudier, à savoir l'impact du changement. Pour cela, nous avons extrait les métriques sur les deux systèmes de l'étude (PRECI et BOAP), à partir des outils d'extraction de métriques précités. En parallèle, le questionnaire sur le calcul d'impact (section 5.1.4) dûment rempli par les quatre sujets est analysé. Les résultats obtenus pour le calcul des métriques et des impacts des changements sont présentés dans les sections suivantes.

5.2.1 Calcul des métriques

En se basant sur les résultats de l'extraction des métriques pour les deux systèmes, nous avons pu établir des statistiques descriptives de la distribution des métriques collectées.

Pour le système PRECI :

Métriques	Min	Max	Mean	Median	St.Dev.
RFC	0	882	43.24	17.5	86.18
MPC	0	695	24.55	3.5	66.00
CBOU	0	19	3.03	1	3.97
CBOIU	0	26	2.93	1	4.81
CBO	0	34	5.46	3	6.31
CBO'	0	34	5.28	3	6.36
DIT	0	2	0.61	1	0.63
NOC	0	10	0.2	0	1.21
DAC	0	24	1.47	0	3.22
DAC'	0	11	0.87	0	1.39
NOC*	0	10	0.22	0	1.24
WMC	1	187	13.97	7	23.56
ACAIC	0	0	0	0	0
ACMIC	0	0	0	0	0
DCAEC	0	0	0	0	0
DCMEC	0	0	0	0	0
OCAEC	0	63	1.36	0	5.99
OCAIC	0	24	1.36	0	3.22
OCMEC	0	95	2.24	0	10.80
OCMIC	0	147	2.24	0	11.80
AMMIC	0	5	0.43	0	1.10
OMMIC	0	695	23.93	2	65.92
DMMEC	0	23	0.43	0	2.59
OMMEC	0	599	24.11	2	71.41

Tableau 7 : Statistiques descriptives des métriques OO analysées (système PRECI)

Pour le système BOAP :

Métriques	Min	Max	Mean	Median	St.Dev.
RFC	0	592	43.83	30	49.06
MPC	0	555	12.43	3	34.87
CBOU	0	38	3.55	2	4.39
CBOIU	0	58	3.49	1	7.86
CBO	0	65	6.96	4	8.92
CBO'	0	64	6.2	3	8.94
DIT	0	4	1.37	1	1.23
NOC	0	17	0.58	0	2.24
DAC	0	11	0.77	0	1.49
DAC'	0	11	0.67	0	1.26
NOC*	0	90	1.2	0	6.65
WMC	0	68	8.23	5	8.67
ACAIC	0	4	0.12	0	0.49
ACMIC	0	7	0.17	0	0.70
DCAEC	0	30	0.12	0	1.61
DCMEC	0	35	0.17	0	2.23
OCAEC	0	18	0.63	0	1.83
OCAIC	0	11	0.63	0	1.29
OCMEC	0	119	1.83	0	8.08
OCMIC	0	15	1.83	1	2.76
AMMIC	0	42	1.99	0	4.37
OMMIC	0	555	10.46	2	34.11
DMMEC	0	230	1.97	0	12.85
OMMEC	0	273	10.48	1	30.01

Tableau 8 : Statistiques descriptives des métriques OO analysées (système BOAP)

D'après les deux tableaux, nous constatons que la hiérarchie d'héritage est assez plate. En effet, la valeur de la médiane de la métrique DIT pour les deux systèmes est égale à 1. Ceci suggère que la majorité des classes tendent à être prés de la racine de l'arbre d'héritage. De plus, les métriques : NOC et NOC* ont des médianes identiques, égale à 0. Cela signifie que les classes ont peu d'enfants immédiats. En général, cela s'explique par l'utilisation de façon limitée du principe d'héritage par les concepteurs des deux systèmes. Une telle remarque est faite par d'autres chercheurs notamment par Chaumun *et al.*[26] et Chidamber et kemerer [22].

D'autre part, il y a beaucoup d'interaction entre les classes, aussi bien dans le système BOAP que dans le système PRECI. En comparant les valeurs maximales des deux métriques CBO et CBO' pour chaque système, on constate qu'elles sont approximativement identiques. On peut donc conclure, que la majorité des interactions sont faites en dehors de l'héritage.

Par ailleurs, le couplage d'importation (import coupling) à partir des autres classes, exprimé par les métriques OCAIC, OCMIC et OMMIC est présent dans les deux systèmes. En effet, plus l'interaction est grande entre une classe et les autres classes (ni ancêtres ni descendants), plus

cette classe est sensible aux changements apportés dans les autres classes. Par conséquent, la maintenance de cette classe est difficile.

De façon semblable, le couplage d'exportation (export coupling) exprimé par les métriques (OCMEC), (OMMEC) et (OCAEC) est présent dans les deux systèmes. En fait, plus le degré d'interaction est intense entre une classe et les autres classes, plus ces dernières sont sensibles aux changements apportés à la première. Par conséquent, la maintenance de cette classe est difficile à cause des dépendances. De plus, on remarque que le couplage via les ancêtres et les descendants ne se présente pas pour le système PRECI, ce qui n'est pas le cas pour le système BOAP. En particulier, l'interaction classe-attribut et classe-méthode dans les deux sens (import coupling et export coupling). Cependant, le couplage via l'interaction méthode-méthode parmi les descendants et les ancêtres est présent dans les deux systèmes.

5.2.2 Calcul d'impact

La synthèse des résultats relatifs au questionnaire fourni à l'équipe du CRIM, portant sur l'impact du changement est présentée dans les tableaux suivants. Ces derniers montrent les occurrences de chaque type d'impact dans le système.

Pour le système PRECI, les résultats sont décrits dans le tableau suivant :

Types de changements	Total de classes	Impact Faible (IF)	Impact Nul (IN)	Impact Moyen (IM)	Impact Grand (IG)
Changement de l'implémentation d'une méthode	39	32	5	2	0
Changement de la portée d'une méthode de publique à privée	38	13	6	17	2
Changement de la signature d'une méthode	0	0	0	0	0
Suppression d'une classe	0	0	0	0	0
Suppression d'une méthode	32	5	1	11	15
Changement du type d'un attribut	20	16	2	2	0
Changement de la portée d'un attribut de publique à privée	30	5	1	22	2
TOTAL	159	71	15	54	19

Tableau 9 : Occurrences des changements pour le système PRECI

Pour le système BOAP, les résultats sont récapitulés comme suit :

Types de changements	Total de classes	Impact Faible (IF)	Impact Nul (IN)	Impact Moyen (IM)	Impact Grand (IG
Changement de l'implémentation d'une méthode	49	3	5	10	31
Changement de la portée d'une méthode de publique à privée	0	0	0	0	0
Changement de la signature d'une méthode	15	10	0	3	2
Suppression d'une classe	1	0	1	0	0
Suppression d'une méthode	2	0	0	1	1
Changement du type d'un attribut	1	0	0	1	0
Changement de la portée d'un attribut de publique à privée	0	0	0	0	0
TOTAL	68	13	6	15	34

Tableau 10 : Occurrences des changements pour le système BOAP

Des deux tableaux sur les occurrences des changements, nous constatons que le changement relatif à la suppression de classes se présente rarement (système BOAP) voire jamais (système PRECI). Nous pouvons donc conclure, que l'architecture de chacun des deux systèmes est stable et ne subit pas de changement.

D'un autre côté, le système PRECI présente une bonne distribution de données des impacts des changements. En effet, 53 classes de l'ensemble des classes du système (160) ont connu au moins un des types de changements ; il s'agit d'un total de 159 occurrences. Cependant, les données relatives à l'impact du changement pour le système BOAP sont faibles. Seulement 51 classes de l'ensemble des classes constituant le système (430) ont subi au moins un des types de changements. En effet, le système BOAP a connu un total de 68 occurrences.

Ainsi, en raison de cette carence de données, nous avons préféré mixer les deux résultats dans le but de construire des modèles significatifs. Dans le reste du chapitre, nous notons par système mixte la combinaison des deux systèmes de l'étude (BOAP et PRECI). Les occurrences des changements du système mixte sont comme suit :

Types de changements	Total des occurrences des changements
Changement de l'implémentation d'une méthode	88
Changement de la portée d'une méthode de publique à privée	38
Changement de la signature d'une méthode	15
Suppression d'une classe	1
Suppression d'une méthode	34
Changement du type d'un attribut	21
Changement de la portée d'un attribut de publique à privée	30
TOTAL	227

Tableau 11 : Occurrences des changements pour le système mixte

5.3 Construction des modèles d'évaluation de l'impact du changement

La construction des modèles d'évaluation vise à souligner la relation entre les propriétés de conception orientée objet dont l'héritage et le couplage, et la caractéristique de qualité qui est dans notre cas l'impact du changement. Les notions de variables indépendantes et variables dépendantes sont beaucoup utilisées dans de telles expériences. Les variables indépendantes sont celles qui sont manipulées ou explicatives. Les variables dépendantes sont mesurées et expliquées.

Pour analyser les modèles, nous avons utilisé le logiciel Open Source "*Weka*" (Waikato Environment for Knowledge Analysis). C'est une collection d'algorithmes d'apprentissage pour la fouille de données (Data Mining) applicable sur un ensemble de données [69]. Nous avons choisi les algorithmes d'apprentissage J48, JRip et IBK [71] pour analyser nos modèles.

Algorithme d'apprentissage J48 : La méthode J48 est inspirée de ID3 de J. Ross Quinlan. C'est un programme pour représenter les règles de classification sous la forme d'un arbre de décision à partir d'un ensemble d'exemples. Les arbres de décisions représentent une approche supervisée de classification. Un arbre de décision est une structure simple où chaque nœud correspond à un attribut et chaque arc à la valeur possible de cet attribut. Les nœuds non terminaux de l'arbre représentent les tests sur un ou plusieurs attributs et les nœuds terminaux

reflètent les résultats de décision; la valeur attendue de l'attribut pour les enregistrements décrits par le chemin de la racine jusqu'à ce nœud terminal.

Algorithme d'apprentissage IBK : Les statisticiens ont analysé les arrangements de "K-nearest-neighbor" au début des années 50. Par la suite, au début des années 60, elle a été adoptée comme une approche de classification. Le classifieur "K-nearest neighbor instance–based learner" normalise les attributs par défaut et choisit la valeur appropriée de K. Dans notre cas, la valeur de K est égale à 1, i.e. l'algorithme cherche parmi les cas existants pour sélectionner la meilleure valeur. Dans l'apprentissage à base de cas, les exemples du jeu sont stockés, et une fonction de distance est utilisée pour déterminer quel membre de l'ensemble du jeu est proche de l'instance de test. Une fois la plus proche instance du jeu localisée, sa classe est prédite pour l'instance de test.

Algorithme d'apprentissage Jrip : Le classifieur Jrip implémente un "rule learner" pour produire une erreur de réduction (RIPPER), qui est proposé par William W.Cohen comme une version optimisée de IREP (Incremental Reduced Error Pruning). Il est considéré parmi les classifieurs les plus utilisés, pour son raisonnement et son principe qui se rapprochent de ceux des experts humains. En outre, il permet à partir d'un ensemble d'exemples de générer des règles compréhensibles et faciles à analyser. Deux types de variables sont utilisés; des variables indépendantes et une seule variable dépendante.

Pratiquement, un algorithme d'apprentissage divise les attributs continus, variables indépendantes (dans notre cas les métriques de conception OO), pour trouver le meilleur seuil parmi un ensemble de jeux de données. En fait, il vise à les classifier sur la variable dépendante (dans cette étude c'est l'impact du changement). Le tableau suivant présente un exemple de données d'entrée " Input Data" nécessaires pour la construction du modèle d'évaluation.

Classes Du système	**Métriques de conception**						**Type d'impact pour Chang1**
	DIT	**NOC**	**CBO**	**RFC**	**OMMEC**	**...........**	
ClassdependencePanel.java	2	0	17	108	5		IL
dbExpression.java	2	0	15	10	2		IF
gui.BlobAnalysisPanel.java	0	0	1	15	9		IN
Compositecomponent.java	1	1	2	15	2		IM
....	...	...	...	...	....	......	....

Tableau 12 : Variables indépendantes et variables dépendantes

Pour utiliser ces données dans les algorithmes d'apprentissage, il faut convertir ces informations en fichier de données. Un exemplaire de ce type de fichier est présenté en annexe (Annexe 5).

5.4 Analyse et interprétations

Pour construire nos modèles de données d'évaluation de l'impact du changement, nous avons procédé par le regroupement des données relatives aux métriques et celles des impacts des changements. De telle sorte qu'une ligne possède le nom de la classe java, le calcul des métriques et le calcul d'impact pour chaque changement qui se présente pour cette classe.

Tel que cité dans la section 5.2.2, nous avons décidé de travailler sur le système mixte (union des résultats des deux systèmes BOAP et PRECI) afin de pouvoir construire des modèles de classification significatifs. Ensuite, nous avons conçu un nouveau modèle d'impact du changement, dérivé du modèle statique qualitatif (section 3.2.2), avec deux types d'impact seulement (impact faible et impact grand) pour avoir un bon échantillon de données. Il est important de noter que le premier est obtenu par l'union deux à deux des quatre types d'impact du deuxième. Le modèle avec deux types d'impact est défini comme suit :

- Impact Faible (IF) : le changement peut avoir ou ne pas avoir un impact local (effet interne) et n'a pas d'impact sur les autres classes du système (pas d'effet externe ni direct ni indirect). Obtenu par la concaténation des deux types d'impact nul (IN) et faible (IF) du modèle d'impact initial.
- Impact Grand (IG) : le changement peut avoir ou ne pas avoir un impact dans la classe changée. Cependant, il a un impact sur le reste du système (effet externe direct et effet externe indirect). Obtenu via la concaténation des deux types d'impact du modèle d'impact initial dont impact grand (IG) et impact moyen (IM).

Dans les sections suivantes nous présentons les modèles construits pour le système mixte avec les deux modèles d'impact (quatre types d'impact et deux types d'impact), ainsi que l'interprétation des résultats.

5.4.1 Système mixte avec modèle à deux types d'impact

5.4.1.1 Changement1 : changement de l'implémentation d'une méthode

a) Jeu de métriques : CBO, CBO', RFC, OCMEC, DAC', NOC, OCAEC

Nombre d'instances : 88, Algorithme : JRIP, Taux de succès : 0.56

Règles : (CBO >= 22) => chang1=IG (13.0/3.0)

(DAC' <= 0) and (RFC >= 27) => chang1=IG (13.0/3.0)

=> chang1=IF (62.0/23.0)

La règle exprimée par (CBO >= 22) => chang1=IG, *signifie que plus le couplage entre objets est grand, supérieur ou égal à 22, plus l'impact du changement de l'implémentation d'une méthode est grand.* En effet, si une classe est couplée à beaucoup d'autres classes, alors l'interdépendance est forte entre elles. Par la suite, un changement de l'implémentation de la méthode qui est beaucoup utilisée par les autres classes du système peut se propager et atteindre toutes ces classes. Par conséquent, l'effort nécessaire pour accomplir le changement est grand.

b) Jeu de métriques : DIT, NOC, NOC*

Nombre d'instances : 88, Algorithme : J48, Taux de succès : 0.78

Règles : DIT <= 1 : IF (81.0/36.0)

DIT > 1 : IG (7.0)

La règle exprimée par DIT> 1 : IG, *permet de dire que si la profondeur de l'arbre d'héritage est supérieure à 1 (2 ou plus), alors l'impact du changement de l'implémentation d'une méthode est grand.* En effet, une classe avec un DIT grand ne se trouve pas à la racine de l'arbre d'héritage, elle est plus spécialisée et hérite de toutes ses super-classes. On peut donc déduire que, plus une classe est profonde dans l'arbre d'héritage, plus le nombre de méthodes et d'attributs dont elle hérite est grand. Ainsi, un changement de l'implémentation d'une méthode qui se trouve à la racine peut se propager vers toute l'arborescence engendrant des efforts supplémentaires pour accomplir le changement.

5.4.1.2 Changement2 : changement de la portée d'une méthode de publique à privée

a) Jeu de métriques: CBO, CBO', RFC, DIT, NOC

Nombre d'instances : 38, Algorithme : J48, Taux de succès : 0.75

Règles : CBO' <= 2 : IF (8.0)

CBO' > 2 : IG (30.0/11.0)

La règle exprimée par CBO' > 2 : IG, *signifie que plus le couplage entre les classes du système en excluant le couplage à travers l'héritage est supérieur à 2, plus l'impact du changement de la portée d'une méthode de publique à privée est grand.* En effet, une méthode publique est facilement accessible par toutes les classes du système. Une fois privée, elle n'est plus visible de l'extérieur et son utilisation est limitée à l'intérieur de sa classe. Les autres classes qui l'utilisaient avant ne peuvent y accéder maintenant que par ses méthodes d'accès (Set( ) et Get()). D'un autre côté, la règle exprimée par CBO' <= 2 : IF, *désigne que si le couplage entre les classes en excluant la relation d'héritage est inférieur ou égal à 2, alors l'impact du changement de la portée d'une méthode de publique à privée est faible.* Cela s'explique par la faible interaction entre les classes du système, qui se répercute sur l'effort nécessaire pour faire le changement.

b) Jeu de métriques : DIT, NOC, OCAEC, OCMEC, OMMEC

Nombre d'instances : 38, Algorithme : J48, Taux succès : 0.66

Règles : NOC <= 0

| OMMEC <= 17 : IF (18.0/3.0)

| OMMEC > 17 : IG (16.0/4.0)

NOC > 0 : IG (4.0)

La règle exprimée par (NOC <=0 et OMMEC >17) : IG, *signifie que si le nombre d'enfants qu'une classe possède est inférieur ou égale à zéro (pas d'enfants) et le nombre de classes, qui sont ni ancêtres ni descendants, ayant une interaction de type méthode-méthode avec un couplage d'exportation est supérieur à 17, alors l'impact du changement est grand.* En effet, quand un nombre important de méthodes définies dans d'autres classes utilisent ou dépendent des méthodes d'une classe particulière, un changement de la portée d'une méthode de publique à privée limitera l'utilisation de cette dernière en dehors de la classe où elle est définie. Par conséquent, l'effort nécessaire pour accomplir le changement est grand, car cette classe satisfait les exigences des méthodes des classes qui n'ont aucun lien d'héritage avec la classe changée.

c) Jeu de métriques : DIT, NOC, NOC*

Nombre d'instances : 38, Algorithme : J48, Taux de succès : 0.78

Règles : NOC <= 0 : IF (34.0/15.0)
NOC > 0 : IG (4.0)

Si le nombre d'enfants qu'une classe possède est supérieur à 0, alors l'impact est grand (NOC > 0 : IG). En effet, si au moins une classe ou plus héritent d'une classe, alors un changement dans la super-classe peut avoir un impact sur toute l'arborescence et causer par la suite la propagation de modification dans toute la descendance. Alors qu'une classe qui n'a pas d'enfants, se trouve à la fin de l'arbre d'héritage ou une classe indépendante, un changement à ce niveau n'affecte pas les descendants puisqu'ils n'existent pas.

5.4.1.3 Changement5 : suppression d'une méthode

Jeu de métriques : DIT, NOC, NOC*

Nombre d'instances : 34, Algorithme : IBK, Taux de succès : 0.82

Le taux le plus élevé est celui donné par l'algorithme IBK. Cependant, on ne peut rien conclure, car aucune connaissance n'est produite.

5.4.1.4 Changement6 : changement du type d'un attribut

Jeu de métriques : OMMEC, DMMEC, OCMEC, OCAEC, DCMEC, DCAEC

Nombre d'instances : 21, Algorithme : JRIP, Taux de succès : 0.95

Règles : (DMMEC >= 19) => chang6=IG (2.0/0.0)
=> chang6=IF (19.0/1.0)

La règle exprimée par DMMEC > 19 => chang6= IG, implique que lorsque l'interaction de type méthode-méthode est supérieure à 19 entre une classe et ses descendants, alors l'impact est grand. En effet, pour un changement dans une classe de base qui possède beaucoup de méthodes, l'impact atteint en plus de ses enfants tous ses descendants par propagation de l'effet du changement.

5.4.1.5 Changement7 : changement de la portée d'un attribut de publique à privée

Jeu de métriques : RFC, MPC, CBO', DAC, DAC', NOC, OMMEC, ACMIC, OCAIC

Nombre d'instances : 30, Algorithme : IBK, Taux de succès : 0.62

Le taux le plus élevé est celui donné par l'algorithme IBK. Cependant, on ne peut rien conclure, car aucune connaissance n'est produite.

5.4.1.6 Synthèse des résultats

L'étude menée avec le système mixte avec modèle à deux types d'impact a concernée principalement les changements suivants :

- Changement de l'implémentation d'une méthode ;
- Changement de la portée d'une méthode de publique vers privée ;
- Suppression d'une méthode ;
- Changement du type d'un attribut ;
- Changement de la portée d'un attribut de publique vers privée.

Cependant, les changements relatifs à la suppression d'une classe et le changement de la signature d'une méthode n'ont pas donné de résultats significatifs. En fait, ceci peut être attribué aux faibles instances de données concernant l'impact du changement.

D'après les résultats de cette étude, nous pouvons dire le changement de l'implémentation d'une méthode et-ou le changement de la portée de publique à privée d'une méthode, qui est utilisée par beaucoup de classes dans le système, va sûrement provoquer des modifications supplémentaires dans ces classes. Ainsi, plus le nombre de clients d'une classe est grand, plus l'impact est grand. Le changement du type d'un attribut influence grandement les classes liées à la classe changée par le principe d'héritage, en particulier les descendants. D'un autre côté, les types de changements relatifs à la suppression d'une méthode et au changement de la portée d'un attribut de publique vers privée ont présenté un bon taux de succès, sauf qu'on ne peut rien décider sur l'impact de ces types de changements.

En résumé, nos deux hypothèses préétablies sur le couplage et l'héritage sont validées par les résultats trouvés. Nous pouvons donc affirmer qu'en général, plus l'interaction entre les classes du système est grande, plus l'impact du changement sur le système est grand, et par conséquent le coût du changement est grand. De façon semblable, plus la classe est profonde dans l'arbre d'héritage, plus l'impact du changement est grand, et par la suite le coût du changement est grand.

Par ailleurs, les métriques d'héritage comme DIT et NOC représentent de bons indicateurs d'impact du changement et les métriques DMMEC, OMMEC, CBO et CBO' relatives au principe de couplage sont de même de bons indicateurs d'impact du changement.

5.4.2 Système mixte avec modèle à quatre types d'impact

5.4.2.1 Changement5 : suppression d'une méthode

a) Jeu de métriques: CBO, CBO', RFC, DIT, NOC

Nombre d'instances : 34, Algorithme : J48, Taux de succès : 0.79

Règles : DIT <= 0

| RFC <= 18

| | CBO <= 6 : IN (2.0/1.0)

| | CBO > 6 : IG (5.0)

| RFC > 18 : IM (15.0/4.0)

DIT > 0 : IG (12.0/1.0)

La règle exprimée par (DIT <=0, RFC <=18 et CBO>6) : IG, *permet de dire que : si la profondeur de la classe dans l'arbre d'héritage est inférieure ou égale à zéro et le nombre de réponse d'une classe (nombre de méthodes pouvant être appelées en réponse à un message) est inférieur à 18 et le nombre de classes couplées à la classe changée est supérieur à 6, alors l'impact est grand.* Cela est évident, puisque le couplage excessif entre une classe et les autres classes du système (non liées par la relation d'héritage) entraîne forcement une dépendance entre elles, et par la suite une grande influence du changement sur le reste du système.

b) Jeu de métriques : DIT, NOC, NOC*

Nombre d'instances : 34, Algorithme : J48, Taux de succès : 0.73

Règles : DIT <= 0 : IM (22.0/10.0)

DIT > 0 : IG (12.0/1.0)

Plus le DIT est grand, plus l'impact augmente (DIT > 0 : IG). Ce résultat confirme celui trouvé pour le système mixte avec le modèle à deux types d'impact.

5.4.2.2 Changement6 : changement du type d'un attribut

Jeu de métriques : OMMEC, DMMEC, OCMEC, OCAEC, DCMEC, DCAEC

Nombre d'instances : 21, Algorithme : J48, Taux de succès : 0.90

Règles : DMMEC <= 0 : IF (19.0/3.0)

DMMEC > 0 : IM (2.0)

La règle exprimée par DMMEC > 0 : IM, signifie que si le couplage via l'interaction méthode-méthode entre la classe et ses descendants est supérieur à 0, alors l'impact est moyen. Cela

s'explique par le fait que le changement du type d'un attribut d'une classe atteint les classes qui lui sont directement liées par la relation d'héritage.

5.4.2.3 Synthèse des résultats

L'étude menée avec le système mixte avec modèle à quatre types d'impact a touchée principalement les changements suivants :

- Suppression d'une méthode ;
- Changement du type d'un attribut ;

Cependant, les autres changements n'ont pas donné de résultats significatifs. En effet, ces changements n'ont pas présenté un taux de succès important.

Il est important de noter que le changement du type d'un attribut influence considérablement le système, en particulier les classes liées directement à la classe changée via la relation d'héritage. De plus, la suppression d'une méthode a un impact grand et peut se propager sur tout le système. En effet, une méthode supprimée n'est plus disponible pour sa classe, ses clients et ses sous-classes. Plus encore, si cette méthode répond aux spécifications de plusieurs classes, alors ces dernières doivent subir des changements importants afin de garder le système cohérent et de ne pas bloquer son traitement.

D'autre part, la métrique d'héritage DIT et les métriques de couplage comme CBO, RFC et DMMEC représentent de bons facteurs de mesure de l'impact du changement. En général, plus une classe est couplée à plusieurs autres classes, plus l'impact est grand. De façon similaire, plus une classe est profonde dans l'arbre d'héritage, plus l'impact est grand. Ainsi, nos deux hypothèses sont validées.

5.4.3 Système PRECI avec modèle à deux types d'impact

Tel que déjà cité, le système PRECI présente une bonne répartition de données relatives au calcul des impacts des changements par rapport au système BOAP. C'est pourquoi dans cette section nous voulons explorer le système PRECI de façon individuel en utilisant le modèle à deux types d'impact.

5.4.3.1 Changement1 : changement de l'implémentation d'une méthode

Jeu de métriques : CBO, MPC, DMMEC, OMMEC, OCAIC, DAC, DAC'

Nombre d'instances : 39, Algorithme : IBK, Taux de succès : 1.00

Nombre d'instances : 39, Algorithme : JRIP, Taux de succès : 0.95

Règles : (CBO >= 32) => chang1=IG (2.0/0.0)

=>chang1=IF (37.0/0.

Si le couplage entre objets est supérieur à 32, alors l'impact du changement de l'implémentation de la méthode est grand. Il apparaît évident qu'une classe avec un fort couplage une fois changée, avec la forte interdépendance entre la classe et ses clients, engendre des efforts supplémentaires pour accomplir le changement. En outre, l'effet du changement peut se propager et atteindre le reste du système.

5.4.3.2 Changement2 : changement de la portée d'une méthode de publique à privée

Jeu de métriques : OCAEC, OCMEC, DIT, NOC

Nombre d'instances : 38, Algorithme : JRip, Taux de succès : 0.69

Règles : (OCMEC >= 2) => chang2=IG (16.0/4.0)

=> chang2=IF (22.0/7.0)

De cette règle OCMEC >= 2 => chang2=IG, on peut conclure que si le nombre d'interaction de type classe-méthode est supérieur à 2, alors l'impact du changement de la portée d'une méthode de publique à privée est grand. En effet, une méthode déclarée publique peut être utilisée par toutes les classes sans restriction. Une fois privée, son utilisation est limitée. Ainsi, des changements supplémentaires doivent être apportés aux classes clientes (qui ne sont ni descendants ni ancêtres de cette classe) afin de résoudre le changement de façon saine.

5.4.3.3 Changement6 : changement du type d'un attribut

Jeu de métriques : CBO, MPC, DMMEC, OMMEC, OCAIC, DAC, DAC'

Nombre d'instances : 20, Algorithme : IBK, Taux de succès : 0.72

Le taux le plus élevé est celui donné par l'algorithme IBK. Cependant, on ne peut rien conclure, car aucune connaissance n'est produite.

5.4.3.4 Changement7 : changement de la portée d'un attribut de publique à privée

Jeu de métriques : CBO, MPC, DMMEC, OMMEC, OCAIC, DAC, DAC'

Nombre d'instances : 30, Algorithme : IBK, Taux de succès : 0.56

Le taux le plus élevé est celui donné par l'algorithme IBK. Cependant, on ne peut rien conclure, car aucune connaissance n'est produite.

5.4.3.5 Synthèse des résultats

L'étude menée avec le système PRECI avec modèle à deux types d'impact a concernée principalement les changements suivants :

- Changement de l'implémentation d'une méthode ;
- Changement de la portée d'une méthode de publique vers privée ;
- Changement du type d'un attribut ;
- Changement de la portée d'un attribut de publique vers privé.

Ces deux derniers changements ont présenté un taux de succès important, cependant aucune décision ne peut être faite sur l'impact de ces types de changements. Le changement relatif à la suppression d'une méthode n'a pas donné de résultats significatifs (faible taux de succès). D'autre part, les deux changements relatifs à la suppression d'une classe et le changement de la signature d'une méthode n'ont pas fait l'objet de l'étude puisque le nombre d'instances de données pour chaque type de changement est nul.

En résumé, pour le système PRECI pris tout seul, nous remarquons que le changement de l'implémentation d'une méthode et le changement de la portée d'une méthode de publique à privée ont un impact grand. Par ailleurs, les métriques de couplage comme CBO et OCMEC représentent de bons indicateurs d'impact du changement. Seule l'hypothèse relative au couplage est validée pour ce système. Nous pouvons donc conclure pour le système PRECI que plus le couplage entre les classes du système est grand, plus l'impact du changement est grand. Par conséquent, le coût du changement est grand et la maintenance sera coûteuse.

5.4.4 Synthèse de l'expérience

Dans un premier temps, nous avons construit les modèles avec deux et quatre types d'impact pour le système mixte (PRECI et BOAP). Par la suite, nous avons conduit une autre étude sur le

système PRECI avec modèle à deux types d'impact. Les résultats du système mixte avec modèle à deux types d'impact sont plus significatifs que ceux du système PRECI.

D'après les résultats des trois études réalisées, nous pouvons dire que les métriques d'héritage comme DIT et NOC sont de bons indicateurs d'impact du changement. Notre première hypothèse ainsi formulée : « la position de la classe dans l'arbre d'héritage d'un système orienté objet affecte l'impact du changement » est validée sur ces deux systèmes. D'autre part, on peut conclure que les métriques de couplage comme CBO, CBO', RFC, OMMEC, DMMEC et OCMEC sont de bons indicateurs d'impact du changement. Ainsi, notre deuxième hypothèse est aussi validée sur nos deux systèmes : « le degré d'interdépendance entre les classes affecte l'impact du changement ».

Par ailleurs, nous pouvons déduire que le changement de l'implémentation d'une méthode, le changement de la portée d'une méthode de publique vers privée, la suppression d'une méthode et le changement du type d'un attribut sont parmi les changements qui propagent l'effet du changement et affectent le reste du système.

5.5 Conclusion

Dans ce chapitre, nous avons mené une étude empirique avec la collaboration du CRIM. Les données de l'étude proviennent de deux systèmes industriels de tailles différentes fournis par le CRIM. Nous avons aussi bénéficié de l'utilisation de l'environnement BOAP et de l'outil que nous avons développé sous Eclipse pour extraire les métriques de conception orientée objet de l'étude. Une interprétation des résultats du calcul de métriques a été faite. Nous avons pu déduire que le couplage entre les classes (qui ne sont ni descendants ni ancêtres) est fort présent dans les deux systèmes et que les deux systèmes utilisent le principe d'héritage de façon limitée. Une telle constatation est faite par d'autres chercheurs notamment par Chaumun *et al.* [26] et Chidamber et Kemerer [22].

Pour valider nos hypothèses, nous avons utilisé quelques algorithmes d'apprentissage (JRIP, J48 et IBK) afin de construire nos modèles prédictifs. Les données relatives au calcul des impacts des changements des deux systèmes étaient insuffisantes pour mener l'expérience sur chaque système seul. C'est pourquoi nous avons mixé les résultats des deux systèmes de l'étude (BOAP et PRECI) en un seul système que nous avons nommé système mixte. D'un autre côté, nous

avons procédé à la concaténation des types d'impact du modèle d'impact initial, donnant naissance à un nouveau modèle avec deux types d'impact.

Nous avons effectivement construit des modèles prédictifs pour le système mixte en utilisant les modèles d'impact avec deux et quatre types d'impact, et pour le système PRECI tout seul avec le modèle à deux types d'impact. De l'analyse et de l'interprétation des résultats, nous avons pu déduire que les changements comme le changement de l'implémentation d'une méthode, la suppression d'une méthode, le changement de la portée d'une méthode de publique à privée et le changement du type d'un attribut sont parmi les changements qui affectent considérablement les systèmes en maintenance. D'autre part, les métriques d'héritage comme DIT et NOC, et de couplage comme CBO, CBO', RFC, OMMEC, DMMEC et OCMEC sont de bons indicateurs de l'impact du changement. Pour bien comprendre les relations entre ces mesures de couplage et d'héritage et l'impact du changement, des études similaires sur d'autres systèmes s'avèrent importantes.

Nous avons donc réussi à démontrer pratiquement que, plus l'interaction entre les classes du système est grande, plus l'impact du changement sur le système est grand. Plus la classe est profonde dans l'arbre d'héritage, plus l'impact du changement est grand. Par la suite, le coût du changement est grand et la maintenance est coûteuse.

Chapitre 6 Conclusions et Perspectives

Ce mémoire traite de l'impact du changement des applications à objets en phase de maintenance. Pour ce faire, nous avons effectué une revue de la littérature des travaux de recherche existants dans le domaine de l'analyse d'impact du changement. Suite à cette revue, nous avons proposé notre démarche pour l'analyse de l'impact. Enfin, nous avons mené une expérience pour analyser l'impact de certains changements sur les classes des systèmes orientés objets.

Dans ce chapitre, nous présentons les conclusions de notre mémoire de recherche, basées aussi bien sur les leçons retenues que sur l'expérience menée. Nous avons pu faire ressortir les axes de recherches futures que nous pensons prometteurs dans le domaine de l'ingénierie du logiciel en général et dans le domaine de la maintenance des systèmes orientés objets en particulier.

6.1 Conclusions

Réduire le coût de la maintenance nécessite de se procurer les moyens nécessaires pour l'accomplir. L'analyse de l'impact du changement est l'une des techniques qui a connu de l'importance dans le domaine de la maintenance. Le but de cette technique est de permettre aux gens responsables de la maintenance d'évaluer le coût du changement à priori, i.e., avant d'entamer l'implémentation du changement. Cette évaluation se fait de deux façons : technique et gestion. La première, évaluation technique, se base sur la compréhension du logiciel en modification et sur l'identification du changement, son impact et ses effets de propagation sur tout le système. Suite à cette analyse technique, l'équipe de maintenance est en mesure de conclure sur la faisabilité technique du changement et proposer des solutions optimales qui ne compromettent pas les fonctionnalités du système existant. La deuxième, évaluation du point de vue gestion (ressources humaines et financières), est établie habituellement par les gestionnaires. Ces derniers peuvent choisir parmi les solutions proposées celle qui satisfait les besoins ou décider de réétudier le changement pour des solutions plus sûres.

Les métriques permettent d'évaluer les structures interne et externe des composants d'un logiciel. En effet, un système logiciel orienté objet est construit à l'aide de classes et de leurs relations. La structure interne est définie par les attributs et les méthodes de la classe et la structure externe du système est définie par les relations entre les classes. Ces liens constituent les canaux de communication entre les classes du système et par conséquent les chemins de propagation de l'effet du changement initial.

L'impact du changement a été étudié de différentes manières par différents utilisateurs, en se basant sur un ensemble de concepts : relations, graphe direct, graphe de relations, etc. Notre approche d'analyse d'impact a pour but d'estimer l'impact du changement dans les programmes à objets à base des relations entre les propriétés architecturales des logiciels (le couplage et l'héritage) et les métriques orientées objets. Cette approche a nécessité l'établissement de nouvelles définitions et la conduite de nouvelles expérimentations.

En premier lieu, nous avons mené une étude pour classifier les changements selon les occurrences qui se présentent dans les programmes orientés objets en phase de maintenance. Pour ce faire, un questionnaire est établi. Les résultats montrent que l'ajout et le changement d'implémentation de méthodes sont parmi les changements qui se présentent souvent. L'ajout de classes est le type de changement dont l'occurrence est rare. La suppression de classes, de méthodes et d'attributs, et le changement des noms de méthodes et d'attributs se présentent rarement. Nous avons pu déceler que les structures (arbre d'héritage) des deux systèmes de l'étude sont stables.

Ensuite, nous avons établi de nouvelles définitions allant de la définition de l'impact du changement, la propagation du changement jusqu'à l'établissement du modèle d'impact et des hypothèses à valider. Nous avons défini l'impact comme le degré de difficulté ressentie pour implémenter le changement de façon appropriée. Un changement dans une classe peut influencer les autres classes du système de trois manières qui sont : effet interne, effet externe direct et effet externe indirect. Le modèle d'impact du changement vise à détecter l'étendue de l'impact du changement sur tout le système. Pour cela, quatre types d'impact sont distingués, selon le degré de l'effet de propagation du changement à travers le système.

Par ailleurs, nous avons procédé à l'extraction des métriques relatives aux propriétés architecturales que l'on voulait évaluer; il s'agit du couplage et de l'héritage. Nous avons

bénéficié de l'environnement BOAP offert par le CRIM pour extraire les métriques. Devant la rareté des métriques implémentées dans ce modèle, nous avons pu l'étendre par l'implémentation de nouvelles métriques. Cependant, à cause du problème de modèle présent dans cette plate-forme limitant l'implémentation des métriques relatives au couplage, nous avons développé un outil sous la plate-forme ECLIPSE qui permet d'analyser un programme orienté objet Java et d'extraire les métriques nécessaires. Cette étape s'est avérée importante pour mener notre expérience.

Par la suite, nous avons mené une expérience afin de vérifier la validité de notre approche et faire ressortir les métriques qui sont de bons indicateurs pour l'impact du changement. Pour valider nos hypothèses, nous avons utilisé quelques algorithmes d'apprentissage afin de construire nos modèles prédictifs. Le plus grand avantage des algorithmes d'apprentissage réside dans l'interprétation des résultats. De plus, ils produisent des modèles prédictifs de bonne qualité, comparables aux modèles basés sur l'analyse statistique. Les modèles ont été validés via un processus de validation croisée.

Pour estimer l'impact de certains changements, nous avons établi un deuxième questionnaire. Ce dernier est donné aux participants à l'expérience pour transcrire leurs perceptions sur la complexité d'apporter les changements en se basant sur le modèle d'impact défini. Ces participants sont les développeurs des systèmes étudiés.
Les données relatives au calcul d'impact des deux systèmes étaient insuffisantes pour explorer chaque système tout seul. C'est pourquoi nous avons opté pour mixer les résultats des deux systèmes de l'étude (BOAP et PRECI) en un seul système que nous notons par système mixte. D'un autre côté, nous avons concaténé les types d'impact du modèle d'impact initial pour obtenir un nouveau modèle avec deux types d'impact seulement.

Les résultats de l'expérience sont probants. Du point de vue types de changements, nous pouvons déduire que les changements comme le changement de l'implémentation d'une méthode, la suppression d'une méthode, le changement de la portée d'une méthode de publique vers privée et le changement du type d'un attribut affectent considérablement les systèmes en maintenance. D'autre part, nous pouvons conclure que les métriques de Chidamber et Kemerer relatives à l'héritage (DIT et NOC) et au couplage (CBO, CBO' et RFC) sont de bons indicateurs de l'impact du changement. Plus encore, les métriques de Briand *et al.* relatives aux

métriques de couplage d'exportation classe-méthode (OCMEC) et méthode-méthode (OMMEC et DMMEC) sont aussi de bons indicateurs de l'impact du changement.

Enfin, nous pouvons conclure que plus l'interaction entre les classes du système est grande, plus l'impact du changement sur le système est grand. Plus la classe est profonde dans l'arbre d'héritage, plus l'impact du changement effectué dans les super-classes est grand sur le système. Par la suite, le coût du changement est grand et la maintenance est coûteuse.

6.2 Principales Contributions

En résumé, nos contributions dans ce travail de recherche sur l'impact du changement des programmes à objets sont :

- l'extension du module de métriques de BOAP par l'ajout de nouvelles métriques ;
- le développement d'un nouvel outil sur la plate-forme Eclipse, qui permet d'analyser et d'extraire les métriques des programmes à objets développés en langage Java ;
- la définition d'une nouvelle approche pour mener l'analyse d'impact du changement allant de la classification des changements selon leurs occurrences de se produire dans les systèmes logiciels, la définition des notions d'impact et de propagation d'impact, l'établissement du modèle pour évaluer l'impact du changement, la proposition des hypothèses à valider jusqu'au choix de la méthode de validation ;
- l'expérimentation sur deux systèmes industriels de tailles différentes, puis l'analyse et l'interprétation des résultats.

6.3 Perspectives

Ce mémoire a permis d'ouvrir de nouveaux horizons de recherche. Parmi ces axes, trois nous ont paru intéressants. Le premier axe de recherche est d'étendre cette approche et permettre l'étude de l'impact du changement entre plusieurs versions d'un système. Comme variante de cet axe, on peut aussi penser à généraliser l'approche pour supporter d'autres langages de programmation.

La deuxième voie de recherche est celle de la finalisation de l'outil que nous avons développé pour englober toutes les métriques de conception orientée objet nécessaires pour l'évaluation des différentes caractéristiques de qualité de logiciel. Cette extension s'avère importante et

bénéfique, car avec Eclipse tous les produits sont des extensions que nous pouvons intégrer à n'importe quel projet.

Les outils automatisés sont nécessaires pour évaluer la qualité des logiciels, en fournissant une extraction rapide des métriques. D'autre part, ils permettent de réduire le temps nécessaire pour la collecte de données surtout s'il s'agit de gros systèmes. Grâce à l'outil BOAP, nous avons pu extraire un nombre important de métriques. Cependant, la résolution du problème de modèle présent dans cette plate-forme permettra l'implantation de nouvelles métriques.

Enfin, bien que les résultats obtenus soient encourageants (malgré les inconvénients, notamment la faible distribution de données relatives au calcul d'impact), il reste à généraliser l'approche présentée dans ce mémoire afin de renforcer la pertinence de nos résultats. En effet, mener d'autres expériences avec notre approche serait d'un grand apport au domaine de la maintenance et en particulier au domaine de l'analyse d'impact du changement. Cela contribuera largement à la détection des facteurs qui augmentent le coût de la maintenance, et à procurer aux développeurs les moyens nécessaires pour concevoir des logiciels moins coûteux.

Bibliographie

[1] Abran, A.; Bourque, P.; Brisebois, R.; Côté, V, La Ré ingénierie du Logiciel : un bref tour d'horizon. In Le génie Logiciel, reproduit des Actes de la conférence "Le génie logiciel et ses applications", huitièmes journées internationales (GL95), vol. 38, EC2 & Cie, Paris, La Défense, 1995, Pages 41-47.

[2] F. B. Abreu, Rogério Carapuça, Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. Venice, Italy, October 1993.

[3] Abreu, F. Brito and Carapuça R, Object-Oriented Software Engineering : Measuring and Controlling the Development Process. Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994.

[4] Abreu, F.B., M. Goulao, and R. Esteves, Toward Design Quality Evaluation of Object-Oriented Software Systems, in Proc. 5th International Conference on Software Quality, Austin, Texas, USA, 1995.

[5] M. A. de Almeida, Hakim Lounis, and W.L. Melo, An Investigation on the Use of Machine Learned Models for Estimating Correction Costs". The 20th International Conference on Software Engineering. April 1998, Kyoto, Japan.

[6] G.Antoniol, G,Canfora and A. de Lucia, Estimating the Size of Changes for Evolving Object Oriented Systems: a case study". In proceedings of the 6th International Software Metrics Symposium, Pages 250-258, boca Raton florida, Nov. 1999.

[7] R. S. Arnold and S. A. Bohner, Impact Analysis - Towards A Framework for Comparison. Proceedings of the Conference on Software Maintenance, September 1993, Pages 292-301.

[8] L.J. Arthur. Software Evolution: The Software Maintenance Challenge. John Wiley & sons, 1998.

[9] S. Barbey, A. Strohmeier, The Problematics of Testing Object-Oriented Software. Second Conference on Software Quality Management, Vol. 2, Pages 411- 426, Edinburgh, Scotland, UK, July 26-28 1994.

[10] V. R. Basili, Software Modeling and Measurement: The Goal/ Question/ Metric/ Paradigm. Computer Science Technical Report Series, CS-TR-2956, University of Maryland, College Park, September 1992.

[11] V. R. Basili, L. Briand and W. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators. In IEEE Transaction on Software Engineering, Vol. 22, No. 10, 1996.

[12] J.M.Bieman, B.K.Kang, Cohesion and Reuse in Object Oriented System. In Proceedings of the Symposium on Software Reusability (SSR'95), Pages 259-262, 1995.

[13] S. A. Bohner, Impact Analysis in Software Change Process : A year 2000 perspectives. In International Conference on Software Maintenance, Pages 42-51, 1996.

[14] S. A. Bohner, R. S. Arnold, An Introduction to Software Change Impact Analysis. Software Change Impact Analysis IEEE Computer society, Press Los Alamitos, California, 1996.

[15] Grady Booch, Object-Oriented Analysis and Design with Applications. Second Edition, Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[16] L. C. Briand, P. Devanbu and W. Melo. An Investigation into Coupling Measures for C++. IEEE, International Conference on Software Engineering 1997.

[17] L.C. Briand, John W. Daly, Jurgen Wust: A Unified Framework for Cohesion Measurement in Object-Oriented Systems. 4th International Software Metrics Symposium (METRICS '97) November 05 - 07, 1997.

[18] L.C. Briand, J. Daly, J. Wuest, A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering, 1999.

[19] L. C. Briand, S. Morasca, et V.R. Basili, Defining and Validating Measures for Object-Based High Level Design Metrics. IEEE Transactions on Software Engineering, Vol. 25, No. 5, 1999.

[20] L.C. Briand, Jurgen Wust, Hakim Lounis, Using Coupling Mesurement for Impact Analysis in Object Oriented Systems. IEEE International Conference on Software Maintenance (ICSM), 1999.

[21] S. Chidamber and C. Kemerer, Towards a Metrics Suite for Object-Oriented Design. In Proceedings Object-Oriented Programming, Systems, Languages, and Applications, Phoenix, AZ, Pages 197-211, November 1991.

[22] S. Chidamber and C. Kemerer, A Metrics Suite for Object- Oriented Design. IEEE Transaction on Software Engineering, Vol. 20, No. 6, Pages 476-493, June 1994.

[23] S. R. Chidamber, D. P. Darcy and C. F. Kemerer, Managerial Use of Metrics for Object-Oriented Software. In IEEE Transactions on Software Engineering, Vol. 24, No. 8, pages 629-639, August 1998.

[24] M. A. Chaumun, Change Impact Analysis in Object-Oriented Systems: Conceptual Model and Application on C++. Master's thesis, Université de Montréal, Montréal, Québec, Canada, Nov. 1998.

[25] M. A. Chaumun, H. Kabaili, R. K. Keller and F. Lustman, A Change Impact Model for Changeability Assessment in Object-Oriented Software Systems. In Proceedings of the Third European Working Conference on Software Maintenance and Reengineering, Pages 130-138, Amsterdam, The Netherlands, March 1999.

[26] M. A. Chaumun. Hind Kabaili, Rudolf K.Keller, François Lustman, and Guy St- Denis, Design Properties and Object-Oriented Software Changeability. Fourth European Conference on Software Maintenance and Reengineering, Zurich, Switzerland, Pages 45-54, February 2000.

[27] Chikofsky Elliot J. and Cross, James H, Reverse Engineering and Design Recovery : A Taxonomy. IEEE Software, Vol. 7, No.1, Pages13-17, 1990.

[28] M. Dorfman, R. H. Thayer, Software Engineering. Los Alamitos, IEEE Computer Society Press, 1997.

[29] L. Dupont, Une Boite à Outils pour les Logiciels. Centre de Recherche Informatique de Montréal (CRIM), http://www.crim.ca/rd/Decouvrir/boap.Decouvrir.pdf

[30] Eclipse, www.eclipse.org, http://jmdoudoux.developpez.com/java/eclipse/ http://www.eclipsetotale.com/articles/FAQEclipse.html#Q1

[31] A. El Hachemi, Modèle de Représentation de Code Source BOAP. Rapport Technique, Centre de Recherche Informatique de Montréal (CRIM), Août 2002.

[32] A. El Hachemi, H. Snoussi, H. Lounis, H. Sahraoui, Boite à Outils d'Analyse de Programmes : BOAP. Centre de Recherche Informatique de Montréal (CRIM), Juillet 2001.

[33] E.Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software.1995.

[34] Glass, R. L. and Noiseux, Ronald A., Software Maintenance Guide Book, Prentice-Hall, Inc., 1981.

[35] Jun Han, Supporting Impact Analysis and Change Propagation in Software Engineering Environments. Peninsula School of Computing and Information Technology. Monash University, McMahons Road, Frankston, Vic. 3199, Australia. Octobre 1996.

[36] M.J. Harrold and G. Rothermel, Selecting Regression Testing for Object-Oriented Software. In Proceedings of the International Conference on Software Maintenance (ICSM' 94), Pages 14-25, September 1994.

[37] P. Hsia, A. Gupta, D. kung, J. Peng, and S. Liu, A Study on the Effect of Architecture on Maintainability of Object-Oriented Systems. Proceedings of the Conference on Software Maintenance, 1995, Pages 4-11.

[38] IEEE STD 1219, Standard for Software Maintenance, 1998.

[39] IEEE Std. 610.12, IEEE Standard Glossary of Software Engineering Terminology,1990.

[40] ISO/IEC 12207, Information Technology, Software Life Cycle Process, 1995.

[41] H. Kabaili , Rudolf K. Keller, and François Lustman, Class Cohesion as Predictor of Changeability: An Empirical Study. In l'Objet, 2001.

[42] H. Kabaili, Rudolf K. Keller, and Francois Lustman, Predicting the Changeability of Object-Oriented Software with the Design Metrics. Submitted to Journal of Software and Systems, 2002.

[43] H. Kabaili, Changeabilité des Logiciels Orientés Objets : Propriétés Architecturales et Indicateurs de Qualités. Thèse Doctorat, Université de Montréal, Faculté des arts et des sciences, 2002.

[44] G. A. Kiran, S. Haripriya and P. Jalote, Effect of Object Orientation on Maintainability of Software. In Proceedings of the International Conference on Software Maintenance, Pages 114-121, October 1997.

[45] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, Change Impact Identification in Object-Oriented Software Maintenance. Proceedings of the Conference on Software Maintenance, 1994.

[46] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, Class Firewall, Test Order and Regression Testing for Object-Oriented Programs. In Journal of Object-Oriented Programming, Pages 51-65, 1995.

[47] D. C. Kung, Jerry Gao, Pei Hsia, A Test Strategy for Object-Oriented Programs. Computer Science Engineering Dept., University of Texas at Arlington. March 25, 1997.

[48] Michelle L. Lee, Change Impact Analysis of Object Oriented Software. PhD Thesis, Information Technology and Engineering, George Mason University, December 1998.

[49] W. Li and S. Henry, Maintenance Metrics for the Object-Oriented Paradigm. Proceedings of the First International Software Metrics Symposium, 1993, Pages 52–60.

[50] Li Li and A. Jefferson Offutt, Algorithmic Analysis of the Impact of Changes to Object-Oriented Software. IEEE International Conference on Software Maintenance, 1996, Pages 171-184.

[51] M. Lindvall, Measurement of Change: stable and Change-prone constructs in a commercial C++ System. In Proceedings of the 6th International Software Metrics Symposium, Pages 40-49, Nov.1999.

[52] H.Lounis, W.L.Melo, Identifying and Measuring Coupling on Modular Systems. 8th International Conferenceon Software Technology (ICST97)- 1997.

[53] Joseph P. Loyall and Susan A. Mathisen, Using Dependence Analysis to Support the Software Maintenance Process. Conference on Software Maintenance, September 1993, Pages 282-291.

[54] N. H. Madhavji, Environment Evolution: The Prism Model of Changes. IEEE Transaction on Software Engineering, Vol. 18, No. 5, May 1992, Pages 380-392.

[55] D. S. McCrickard and Gregory D. Abowd, Assessing the Impact of Changes at the Architectural Level: A case study on graphical debuggers. In Proceedings of the International Conference on Software Maintenance (ICSM'96), February 1996.

[56] C. McClure and J. Martin, Software Maintenance : the Problem and Its Solutions. Prentice Hall, 1983.

[57] R. Moreton, A Process Model for Software Maintenance. Journal Information Technology, Volume 5, 1990, Pages100-104.

[58] M. Munro, O. C. Kwon, C. Boldyreff, Survey on a Software Maintenance Support Environment. Technical Report Centre for Software Maintenance, University of Durham, 1998.

[59] S. L. Pfleeger and Shawn A. Bohner, A Framework for Software Maintenance Metrics. In proceedings of the Conference on Software Engineering, May 1990, Pages 320-327.

[60] S.L. Pfleeger, Software Engineering : Theory and Practice. Prentice Hall, 1998.

[61] T. M. Pigoski, Practical Software Maintenance : Best Practices for Managing your Software Investment. John Wiley & Sons, 1997.

[62] M. H. Robert Cantave, Abstractions via un Modèle Générique d'Applications Orientées Objets. Maître ès sciences (M.Sc.), Université de Laval, Août 2001.

[63] H.D. Rombach. Design Measurement: Some Lessons Learned. In IEEE Software, Vol. 7, No. 2, pages 17- 25, 1990.

[64] J. Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[65] W. P. Stevens, G. Myers, L. Constantine, Structured Design. IBM Systems Journal, Vol. 13, No. 2, 1974.

[66] Swebok, Guide to the Software Engineering Body of Knowledge. IEEE Version 2004. http://www.swebok.org/.

[67] R. J. Turver and M. Munro, An Early Impact Analysis Technique for Software Maintenance. Journal of Software Maintenance Research and Practice, Vol. 6, No. 1, 1994.

[68] S. Warshall, A Theorem on Boolean Matrices. J.ACM, Vol.9, No.1, 1962, Pages 11-12.

[69] Weka, Waikato Environment for Knowledge Analysis (WEKA). http://www.cs.waikato.ac.nz

[70] N. Wilde and Ross Huitt, Maintenance Support for Object- Oriented Programs. IEEE Transaction Software Engineering, Vol.18, No. 12, 1992, Pages 1038-1044.

[71] Witten I. H. and Eibe Frank, Data Mining: practical machine learning tools and techniques with Java implementations. San Francisco, Calif., 2000.

[72] S. S. Yau and J. Collofello, Some Stability Measures for Software Maintenance. IEEE Transactions on Software Engineering, Volume SE-6, No. 6, November 1980, Pages 545-552.

[73] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot, Recovering Binary Class Relationships: Putting Icing on the UML Cake. Proceedings of OOPSLA, 2004.

Annexes

Annexe 1 : Liste des métriques de la littérature

Métriques de Chidamber et Kemerer [22]	
Métriques	**Descriptions**
WMC	Weihted Methods per Class : représente la somme des complexités de toutes les méthodes d'une classe.
DIT	Depth of Inheritance : Profondeur de la classe dans l'arbre d'héritage. Dans le cas de l'héritage multiple, le DIT sera le plus long chemin de la classe pour atteindre la racine de l'arbre d'héritage.
NOC	Number Of Children : réfère au nombre de classes immédiatement dérivées d'une classe dans la hiérarchie de classe.
CBO	Coupling Between Object : représente le nombre de classes avec lesquelles une classe est couplée. Deux classes sont dites couplées si les méthodes définies dans l'une utilisent des méthodes ou des variables d'instances de l'autre.
CBO'	Même chose que CBO sauf qu'elle compte le nombre de classes avec lesquelles une classe c est couplée et n'ayant pas une relation d'héritage. Autrement dit, le couplage entre la classe cible et ses ancêtres n'est pas pris en considération puisque les antécédents de la classe changée ne peuvent jamais être affectés.
RFC	Response For a Class : vaut la cardinalité de l'ensemble de réponses d'une classe.
LCOM	Lack of Cohesion in Methods : le nombre de paires de méthodes ne partageant aucune variable d'instance, moins, le nombre de paires de méthodes partageant des variables d'instances de la classe.
Métriques de Li et Henry [49]	
Métriques	**Descriptions**
MPC	Message Passing Coupling : le nombre de messages envoyés d'une classe en direction des autres classes.
DAC	Data Abstraction Coupling: correspond au nombre d'ADT définis dans une classe.
NOM	Number of Methods in a class : correspond au nombre de méthodes locales.
SIZE1	le nombre de points virgules dans une classe.
SIZE2	le nombre d'attributs et de méthodes définis dans une classe.

Métriques de Abreu Brito et Carapuça [2]	
Métriques	**Descriptions**
Design method	Nombre de variables d'instances utilisées. Pourcentage de variables d'instances utilisées.
Design system	Nombre moyen de méthodes par classe. Nombre moyen de variables d'instances par classe.
Size method	Nombre d'instructions exécutables. Nombre de variables d'instances utilisées.
Size class	Nombre de méthodes. Nombre total de variables d'instances.
Size system	Nombre de classes. Nombre total de méthodes. Nombre total de variables d'instances.
Complexity class	Nombre d'enfants Nombre de descendants. Nombre de super-classes directes d'une classe. Nombre de toutes les super-classes directes ou indirectes d'une classe. Nombre de sous-classes directement liées à une classe.
Complexity system	Longueur totale de la chaîne d'héritage.
Reuse class	Pourcentage de méthodes héritées et qui sont redéfinies.

Métriques de Abreu Brito et Carapuça [3]	
Métriques	**Descriptions**
AHF	Attribute Hiding Factor : pourcentage d'attributs cachés.
MHF	Method Hiding Factor : pourcentage de méthodes cachées.
MIF	Method Inheritance Factor : pourcentage de méthodes héritées.
AIF	Attribute Inheritance Factor : pourcentage d'attributs hérités.
PF	Polymorphism Factor : pourcentage de méthodes polymorphes par rapport au nombre total de méthodes potentiellement polymorphes.
CF	Coupling Factor : pourcentage de classes couplées aux autres classes autrement que par l'héritage.

Métriques de Briand *et al.* [16]	
Métriques	**Descriptions**
ACAIC	Ancestor class-attribute import coupling : ACAIC(c) ={a/ a ∈ Ai(c) ∧ T(a) ∈ Ancêtres(c) }. C'est la cardinalité de l'ensemble des attributs implémentés dans la classe c Ai(c) (c'est à dire les attributs non hérités) et dont les types de ces attributs font parti des ancêtres de la classe c.
OCAIC	Others class-attribute import coupling : Nombre d'attributs, qu'une classe particulière possède de type une des classes du système, où ces dernières n'existent ni parmi les descendants ni les ancêtres de cette classe.
DCAEC	Descendents class-attribute export coupling : Nombre de descendants d'une classe particulière qui possèdent un attribut de type cette classe.
OCAEC	Others class-attribute export coupling : Nombre de classes autre que les descendants et les ancêtres de la classe observée et qui ont un attribut de type la classe observée.
ACMIC	Ancestors class-method import coupling : Nombre de paramètres de nouvelles méthodes, que la classe observée possède de type une des classes du système. Ces dernières doivent être l'ensemble des ancêtres de la classe observée.
OCMIC	Others class-method import coupling : Nombre de méthodes n'appartenant ni aux ancêtres et ni aux descendants de la classe observée, et qui possèdent une méthode ayant comme paramètre la classe observée.
DCMEC	Descendants class-method export coupling : Nombre de méthodes existant parmi les descendants d'une classe particulière et qui possèdent un paramètre de type cette classe.
OCMEC	Others class-method export coupling : Nombre de classes n'appartenant ni aux ancêtres ni aux descendants d'une classe particulière, possédant une méthode qui a un paramètre de type cette classe.
AMMIC	Ancestors method-method import coupling : Nombre de classes parentes avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type IC.
OMMIC	Others method-method import coupling : Nombre de classes (autres que les super-classes et les sous-classes) avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type IC.
DMMEC	Descendants method-method export coupling : Nombre de sous-classes avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type EC.
OMMEC	Others method - method export coupling : Nombre de classes (autres que les super-classes et les sous-classes) avec lesquelles une classe a une interaction de type méthode-méthode et un couplage de type EC.

Métriques de Chaumun et Kabaili [26]	
Métriques	**Descriptions**
NOC*	Number of children in sub-tree : le nombre d'enfants dans le sous-arbre d'héritage.
CBO-IUB	CBO is used by : la métrique CBO inclut le couplage en deux sens : classe utilisant la classe cible et classe utilisée par la classe cible. CBO-IUB se réfère aux classes utilisant la classe cible.
CBO-U	CBO Using : la partie du CBO qui fait référence aux classes utilisées par la classe cible. C'est une conséquence du CBO-IUB.

Métriques de Briand *et al.* [20]	
Métriques	**Descriptions**
SIMAS	Nombre de méthodes invoquées directement ou indirectement de la classe D. Prend en considération les invocations statiques seulement.
PIM	Nombre d'invocations de méthodes dans C des méthodes de la classe D. Prend en considération le polymorphisme.
PIMAS	Nombre de méthodes invoquées directement ou indirectement de la classe D. Prend en considération le polymorphisme.
INAG	Couplage indirect, prend en considération la fermeture transitive. En effet, si la classe C possède un attribut de type la classe D et D possède un attribut de type la classe E, alors les classes C et E sont indirectement couplées.

Annexe 2 : liste des métriques OO d'impact du changement [48]

Métriques	Descriptions	Observations
Number of impacted classes	Nombre de classes affectées dans un système. Plus le nombre est petit, plus faible sera l'impact du changement sur le système.	La borne inférieure de cette métrique est le nombre de classes incluses dans le critère de changement (aucun impact sur le reste du système). Sa borne supérieure représente toutes les classes du système (impact sur tout le système).
Percentage of impacted classes	Nombre de classes affectées dans un système divisé par le nombre de classes du système. Plus le résultat est petit, moins d'impact a le changement sur le système.	La borne inférieure est le nombre de classes affectées divisé par le nombre de classes du système. La borne supérieure est égale à 1, i.e., toutes les classes du système sont affectées.
Number of impacted members	La somme de tous les membres affectés de toutes les classes affectées du système.	La borne inférieure est le nombre de membres affectés inclus dans le changement initial. La borne supérieure est le nombre de tous les membres de toutes les classes du système.
Average number of impacted members	La somme de tous les membres affectés divisé par la somme de tous les membres du système.	
Weighted number of impacted members	La somme de tous les membres affectés de toutes les classes affectées du système, auxquels des constantes de poids sont attribuées.	
Average Weighted number of impacted members	C'est le résultat de la métrique Weighted number of impacted members divisé par le nombre total de membres du système.	
Method impact level (MIL)	Mesure l'impact à l'intérieur d'une méthode. Elle considère non seulement les variables et les instructions affectées, mais aussi la taille et la complexité de la méthode elle-même.	Plus la méthode est grande, plus c'est difficile de la comprendre et la modifier, alors la taille a un impact sur la modifiabilité de la méthode. Des programmes petits peuvent être difficile à comprendre et modifier a cause de leurs complexités.
Class impact level (CIL)	Mesure le niveau d'impact à l'intérieur d'une classe. Elle prend en considération les impacts sur les méthodes et les variables, et la contribution de la taille et la complexité de la classe au niveau de l'impact.	Pour les classes de l'arbre d'héritage, la taille peut être mesurée de deux façons : en considérant seulement les membres locaux et en considérant les membres locaux plus tous les membres hérités.
System impact level (SIL)	Mesure l'impact au niveau système.	C'est la somme de tous les niveaux d'impacts des classes plus la complexité du système.

Annexe 3 : Questionnaire sur le recensement des types de changements

Changements au niveau classe :

Types de changements	**Fréquence (F)**		
	Jamais F=0	**Rarement 1<=F<5**	**Souvent F>=5**
Ajout d'une classe			
Suppression d'une classe			
Changement d'une classe Non Abstraite--------------Abstraite			
Changement d'une classe Abstraite ------------------Non Abstraite			
Ajout d'une classe Abstraite			
Ajout d'une classe non Abstraite			
Suppression d'une classe Abstraite			
Suppression d'une classe non Abstraite			
Ajout d'une super-classe			
Suppression d'une super-classe			
Ajout d'une sous-classe			
Suppression d'une sous-classe			
Ajout/suppression d'une relation d'héritage/d'agrégation/d'association			
Ajout/suppression d'une classe et ses relations			
Ajout/suppression d'une classe indépendante			

Changements au niveau attribut :

Types de changements	**Fréquence (F)**		
	Jamais F=0	**Rarement 1<=F<5**	**Souvent F>=5**
Ajout d'un attribut			
Suppression d'un attribut			
Changement de la déclaration d'un attribut			
Changement de valeur			
Changement de type			
Changement de nom			
Changement de la portée d'un attribut			
Publique------------privée			
Publique------------protégée			
Protégée----------- privée			
Protégée-----------publique			
Privée -------------publique			
Privée -------------protégée			
Changement d'un attribut (static/ non- static)			
Static ------------- non-static			
Non-static-------- Static			

Changements au niveau méthode :

Types de changements	**Fréquence (F)**		
	Jamais F=0	**Rarement 1<=F<5**	**Souvent F>=5**
Suppression d'une méthode			
Ajout d'une méthode			
Changement de la signature			
Changement du type de retour			
Changement de nom			
Changement de la liste des paramètres			
Changement du nom d'un paramètre			
Changement de portée d'une méthode			
Publique----------------privée			
Publique----------------protégée			
Protégée----------------privée			
Protégée----------------publique			
Privée ------------------publique			
Privée ------------------protégée			
Changement d'une méthode Static-------------- non Static			
Changement d'une méthode Non-static------------- Static			
Changement d'implémentation d'une méthode			
Ajout / suppression / changement d'appel de fonction			
Ajout/Suppression/ changement d'une structure de contrôle			
Modification de séquence des instructions			
Ajout/suppression d'une séquence d'instructions			
Ajout/ suppression/ modification d'une variable locale			

Annexe 4 : Résultats du questionnaire sur le recensement des types de changements (Systèmes : BOAP et PRECI)

Résultats des changements au niveau classe :

Types de changements	Fréquence (F)		
	Jamais F=0	Rarement 1<=F<5	Souvent F>=5
Ajout d'une classe		4	
Suppression d'une classe	2	2	
Changement d'une classe Non Abstraite--------------Abstraite	4		
Changement d'une classe Abstraite ------------------Non Abstraite	4		
Ajout d'une classe Abstraite	4		
Ajout d'une classe non Abstraite		4	
Suppression d'une classe Abstraite	4		
Suppression d'une classe non Abstraite	1	3	
Ajout d'une super-classe	4		
Suppression d'une super-classe	4		
Ajout d'une sous-classe	3	1	
Suppression d'une sous-classe	3	1	
Ajout/suppression d'une relation d'héritage/d'agrégation/d'association	2	2	
Ajout/suppression d'une classe et ses relations	3	1	
Ajout/suppression d'une classe indépendante	2	2	

Résultats des changements au niveau attribut :

Types de changements	Fréquence (F)		
	Jamais F=0	**Rarement 1<=F<5**	**Souvent F>=5**
Ajout d'un attribut		3	1
Suppression d'un attribut	1	3	
Changement de la valeur d'un attribut	2	1	1
Changement du type d'un attribut		4	
Changement du nom d'un attribut		4	
Changement de portée d'un attribut			
Publique------------privée	2	1	1
Publique------------protégée	3	1	
Protégée ---------- privée	4		
Protégée--- --------publique	4		
Privée --------------publique	3		1
Privée --------------protégée	4		
Changement d'un attribut Static ------------- non–static	4		
Changement d'un attribut Non-static-------- Static	3	1	

Résultats des changements au niveau méthode :

Types de changements	Fréquence (F)		
	Jamais F=0	**Rarement 1<=F<5**	**Souvent F>=5**
Suppression d'une méthode		4	
Ajout d'une méthode		1	3
Changement du type de retour	1	2	1
Changement du nom		4	
Changement de la liste des paramètres		2	2
Changement du nom d'un paramètre		3	1
Changement de portée d'une méthode			
Publique--------------------privée		3	1
Publique--------------------protégée	2	2	
Protégée --------------------privée	4		
Protégée--------------------publique	4		
Privée ----------------------publique	1	2	1
Privée ----------------------protégée	2	2	
Changement d'une méthode Static ------------------ non-static	3	1	
Changement d'une méthode Non-static------------- Static	3	1	
Changement d'implémentation d'une méthode			
Ajout / suppression / changement d'appel de fonction		1	3
Ajout/Suppression/ changement d'une structure de contrôle		1	3
Modification de séquence des instructions		2	2
Ajout/suppression d'une séquence d'instructions		1	3
Ajout/ suppression/ modification d'une variable locale		1	3

Annexe 5 : Fichier de données

Le fichier de données utilisé par les algorithmes d'apprentissage définit les classes et les attributs, et leurs valeurs respectives. Il est utilisé pour décrire l'ensemble des cas constituant le jeu de données à partir duquel les arbres de décisions sont construits et-ou les règles sont produites. Composé de deux parties : la première correspond aux déclarations d'attributs et de classes. Une déclaration débute par le mot clé " ATTRIBUTE", suivi du nom de l'attribut et de son type. Dans notre cas, les premières lignes de déclarations correspondent aux métriques de conception orientée objet, et elles sont toutes de type réel. La dernière déclaration correspond à la classe, avec énumération de l'ensemble des valeurs qu'elle peut prendre séparées par des virgules; il s'agit de : IN pour impact nul, IM pour impact moyen, IG pour impact grand ou IF pour impact faible.

Déclaration des attributs et classes

```
@ATTRIBUTE RFC REAL
@ATTRIBUTE MPC REAL
@ATTRIBUTE CBONA REAL
@ATTRIBUTE DAC REAL
@ATTRIBUTE DACC REAL
@ATTRIBUTE NOC REAL
@ATTRIBUTE OMMEC REAL
@ATTRIBUTE ACMIC REAL
@ATTRIBUTE OCAIC REAL
@ATTRIBUTE Changl {IN, IM, IG, IF}
```

Données relatives aux attributs et classes

```
@DATA
108,64,15,0,0,0,5,0,0,IM
64,37,9,0,0,0,0,0,0,IM
17,8,4,1,1,0,0,0,1,IF
53,41,11,3,3,0,11,0,3,IG
31,12,4,2,2,0,3,0,2,IG
32,18,4,0,0,0,16,0,0,IG
62,34,10,0,0,0,0,0,0,IM
```

Exemple de fichier de données : test.arff

La deuxième partie est constituée des données relatives aux déclarations faites dans l'entête du fichier. Chaque ligne de données décrit un cas en fournissant toutes les valeurs des attributs (métriques de conception) et la classe de ce cas (type d'impact) séparées par des virgules. Les valeurs des attributs doivent apparaître dans le même ordre que les déclarations faites dans l'entête du fichier. Il est essentiel de souligner que l'ordre des valeurs est important tandis que l'ordre des cas n'est pas important.