UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
UNIVERSITE DU QUEBEC A TROIS-RIVIERES

COMME EXIGENCE PARTIELLE
DE LA MAITRISE EN MATHEMATIQUES ET INFORMATIQUE APPLIQUEES

PAR
NICOLAS JOLY

ÉLABORATION D'UN MODELE D'IMPACT DU CHANGEMENT POUR LES APPLICATIONS JAVA

DECEMBRE 2010

PAGE PRÉSENTATION JURY

ÉLABORATION D'UN MODELE D'IMPACT DU CHANGEMENT POUR LES APPLICATIONS JAVA

Nicolas Joly

SOMMAIRE

La maintenance représente une phase très importante du cycle de vie d'un logiciel. Un logiciel peut, en effet, subir durant sa vie un grand nombre de changements pour, entre autres, introduire de nouveaux besoins, l'adapter à un nouvel environnement, corriger des erreurs éventuelles ou tout simplement améliorer ses performances. Ces changements, en particulier dans le cas de logiciels larges et complexes, peuvent avec le temps dégrader leur qualité. Ils peuvent aussi être coûteux à mettre en œuvre. L'analyse de l'impact d'un changement joue un rôle primordial dans ce contexte. Elle permet aux développeurs d'évaluer les effets d'un changement. Elle peut également fournir, quand elle est utilisée de façon prédictive, des informations sur les efforts nécessaires à son implémentation. Dans ce contexte, nous avons élaboré un modèle d'analyse d'impact du changement spécifique aux applications Java. Le modèle a été conçu pour offrir aux développeurs la possibilité de faire de l'analyse d'impact en cascade, avec des règles d'impact qui précisent (le plus possible) quels seront les différents types d'impacts et leur localisation dans le code. Les premières expérimentations effectuées sur plusieurs études de cas simples ont montré que le modèle est capable d'atteindre un taux de précision élevé comparé aux autres modèles prédictifs basés sur le code source disponibles dans la littérature. Le modèle devra, cependant, être évalué à plus grande échelle pour tirer des conclusions finales.

ELABORATION OF A CHANGE IMPACT MODEL FOR JAVA APPLICATIONS

Nicolas Joly

ABSTRACT

Maintenance is a very important part of the life cycle of software. During its life, software can be subject to an important number of changes for, among others, introducing new requirements, adapting the software to a new environment, correcting errors or simply improving its performance. These changes, particularly in the case of large and complex software, can degrade their quality over time. They can also be very expensive to implement. The impact analysis of a change plays an essential role in this context. It allows developers to evaluate the effects of a change. It can also provide, when used in a predictive way, information on the efforts needed for its implementation. In this context, we developed a change impact analysis model specific to Java systems. The model was designed to give developers the ability to make impact analysis taking into account the ripple effect, with impact rules that specify (as possible) what are the different impacts and their location in the code. The first experiments performed using various simple case studies have shown that the model is able to achieve a high accuracy rate compared to other predictive models based on the source code available in the literature. The model should however be evaluated on large scale systems to draw final conclusions.

REMERCIEMENTS

Un Projet de cette taille nécessite beaucoup de travail et surtout l'aide et le support de nombreuses personnes.

Il faut premièrement souligner l'effort continu apporté par mes directeurs de maîtrise Mourad Badri et Linda Badri, professeurs au Département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières. Grâce à leurs expériences et à leurs grandes compétences, ils ont su me maintenir sur la bonne direction et me diriger habilement dans mon cheminement de maîtrise. Vous avez su me donner les mots clés aux bons moments pour surmonter les embûches rencontrées sur mon chemin.

Je tiens aussi à remercier l'ensemble de ma famille qui m'a encouragé tout au long de mon parcours scolaire. Un merci particulier à mon frère Adam pour l'aide qu'il m'a apportée pour une première révision de mon mémoire.

J'aimerais aussi remercier tous mes collègues de maîtrise avec lesquels j'ai échangé au sujet de mon mémoire et qui m'ont permis de pousser au maximum la réflexion sur l'utilisation du modèle présenté. Je n'aurais pu faire autant d'expérimentations sur le modèle sans l'aide, entre autres, d'Emeric Garon et de Michel St-Arnaud.

Un dernier merci à tous les professeurs qui m'ont fourni du travail au cours de ma maîtrise et qui m'ont ainsi soutenu financièrement.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

CHAPITRE 1

INTRODUCTION

Dans le cycle de vie d'un logiciel, la maintenance prend de plus en plus d'ampleur au niveau des coûts et du temps. Les logiciels doivent être en mesure d'évoluer toujours plus rapidement et plus facilement. La maintenabilité est une caractéristique importante des logiciels. Ceci n'est pas un hasard. Le succès d'un logiciel est, entre autres, fortement lié à sa longévité et à ses coûts.

La maintenabilité, c'est la capacité d'un logiciel à évoluer (modification et adaptation) tout en gardant ses qualités. Un logiciel, pendant sa maintenance, peut être modifié pour : corriger des erreurs présentes à la livraison du logiciel et/ou introduites par d'autres activités de maintenance, ajouter des fonctionnalités pour répondre à la demande des clients (nouveaux besoins), s'adapter aux nouvelles réalités de l'informatique qui progressent rapidement (nouvelle plate-forme de déploiement, nouveau protocole de sécurité, nouvelle version de langage, nouveaux standards de qualité ou d'usage, etc.) et éventuellement pour améliorer certains aspects relatifs à sa qualité. La maintenabilité peut avoir indirectement une incidence sur la viabilité d'un logiciel. Elle permet de fournir un indice important sur la qualité du développement du logiciel en lien avec sa maintenance.

Dans les différentes recherches présentées dans ce document, on ne cherche pas à mesurer la maintenabilité. On tente plus à fournir des outils pour l'améliorer et pour soutenir la phase de maintenance. Dans les différentes activités liées à la maintenance, comme il a été mentionné précédemment, le code source d'un logiciel peut être modifié pour diverses raisons.

L'analyse automatique du code, pour supporter certaines activités de la maintenance, est particulièrement nécessaire pour des logiciels dont la taille et la complexité sont de plus en plus grandes. Elle est également nécessaire pour prendre en charge des besoins urgents. Elle est d'autant plus importante, en particulier, lorsque l'équipe de maintenance n'est pas la même que celle qui s'est occupé du développement. L'analyse automatique va permettre de faciliter le processus de compréhension avant

d'entreprendre toute modification et permettre ainsi de réduire certains coûts liés à la maintenance en évaluant ce que la tâche implique en termes de difficulté, de temps et de coûts. On doit être en mesure d'identifier les impacts relatifs à un changement donné et les corrections à apporter avec un moindre coût. Le principal objectif est de faciliter l'évolution du logiciel. Il s'agit d'une tâche primordiale et particulièrement complexe [26].

Notre travail porte sur l'analyse de l'impact des changements, peu importe le type de maintenance (corrective, adaptative ou perfective). L'analyse de l'impact d'un changement permet d'évaluer les implications liées à ce changement. Les changements que peut subir un logiciel (en particulier dans le cas de logiciels larges et complexes) peuvent avec le temps dégrader sa qualité. Ils peuvent aussi être couteux à mettre en œuvre. L'analyse de l'impact d'un changement joue un rôle primordial dans ce contexte. Elle permet effectivement aux développeurs d'évaluer les effets d'un changement. Elle peut également fournir, quand elle est utilisée de façon prédictive, des informations sur les efforts nécessaires à son implémentation.

Nous avons développé un modèle d'analyse d'impact du changement ayant pour objectif d'évaluer les répercussions d'un changement dans le code. Le modèle se veut précis et capable d'effectuer une analyse en cascade pour obtenir une évaluation concrète des répercussions d'un changement. De plus, cette analyse a pour objectif d'outre passer l'aspect évaluation en fournissant des informations pertinentes pouvant supporter son implémentation.

Pour ce faire, nous avons d'abord commencé par étudier les modèles (et travaux d'une manière générale) portant (ou reliés) sur l'analyse d'impact du changement dans la littérature. Cela nous a permis de mieux identifier (et cibler par la suite) les insuffisances que notre modèle doit combler. Cette étude nous a permis d'établir un modèle d'impact du changement basé sur le code. Nous avons ciblé les applications Java. Nous avons aussi focalisé dans notre travail sur l'analyse prédictive que devra offrir le modèle (pour les différents avantages que cela procure). Par la suite, nous avons soumis le modèle à plusieurs évaluations simples (pour le moment) mais néanmoins concrètes et pertinentes. Les résultats obtenus sont prometteurs.

Dans ce document, nous abordons dans un premier temps les travaux portant sur le domaine de la maintenance et plus précisément l'analyse d'impact au chapitre 2. Cela permet d'établir les lignes directrices de notre recherche. Dans un second temps, au chapitre 3, nous présentons en détail un modèle connu dans la littérature (le modèle de Chaumun) et le modèle que nous proposons. Le modèle de Chaumun servira de base de comparaison pour évaluer notre modèle d'analyse de l'impact. Au chapitre 4, nous présentons les nombreuses expérimentations que nous avons effectuées et les résultats obtenus. Le chapitre 5 est dédié à la présentation du prototype supportant notre approche. On termine enfin le mémoire par le chapitre 6 dans lequel nous présentons une conclusion ainsi que quelques perspectives à ce travail.

CHAPITRE 2

REVUE DE LA LITTÉRATURE

2.1 Introduction

Il est reconnu que la maintenance représente un élément important, voire le plus important dans certains cas, de la vie d'un logiciel. Plusieurs travaux (à différents niveaux) ont été réalisés dans le but de mieux comprendre (pour mieux appréhender et améliorer) la maintenabilité d'un logiciel, et de supporter une ou plusieurs étapes du processus de maintenance. Une de ces activités importantes porte sur l'analyse de l'impact des changements.

Dans ce chapitre, nous aborderons dans un premier temps plusieurs travaux reliés à la maintenance d'une manière générale. Puis, nous nous attarderons plus spécifiquement sur l'analyse de l'impact, objet principal de notre travail, en différenciant les notions de post-analyse et d'analyse prédictive. Enfin, nous discuterons des diverses approches liées à l'analyse de l'impact.

2.2 Maintenance

La maintenance fait partie du cycle de vie d'un logiciel. Cette étape prend de plus en plus d'importance (voir figure 1 tirée de [37]) dans la réalité concurrentielle de l'informatique. Les logiciels sont de plus en plus volumineux, doivent accomplir de plus en plus de tâches, sont toujours plus complexes, ont des besoins en constante évolution et doivent être développés rapidement. Elle est donc devenue avec le temps une étape essentielle et très coûteuse [18][11][12]. Dans certains cas, la maintenance peut représenter 70% du coût d'un logiciel [11][37]. Une partie des coûts est liée à la mauvaise gestion de la maintenance, car il est très difficile d'évaluer ces coûts efficacement avant de prendre une décision [13][14][16].

Selon le standard IEEE 1219 [12], la maintenance est définie ainsi:

« La modification d'un produit logiciel après sa livraison pour corriger les erreurs, pour améliorer ses performances ou d'autres attributs, ou pour adapter le produit à un environnement modifié. »

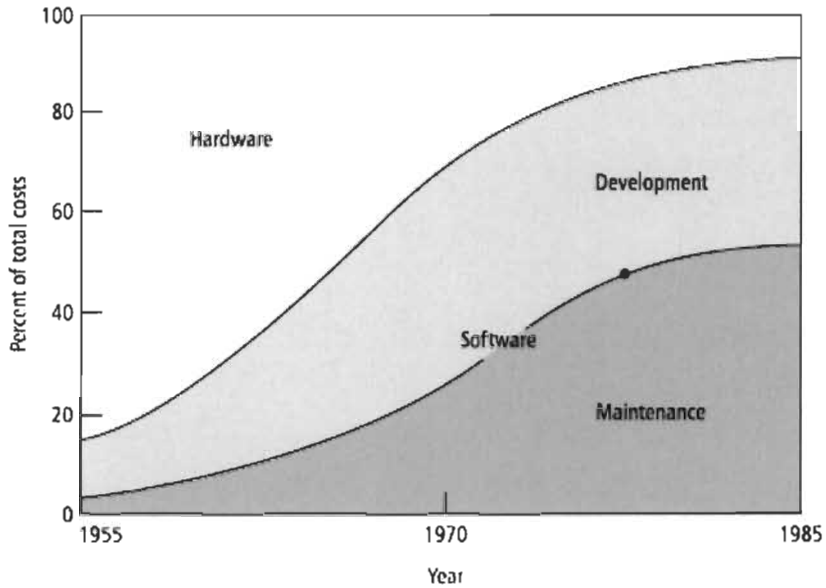

Figure 1 Distribution relative des coûts matériels/logiciels [37].

La maintenance est considérée comme l'ensemble des activités adaptatives, correctives ou perfectives liées au logiciel après son déploiement [12][15]. C'est donc aussi l'ensemble des activités permettant le prolongement de la vie d'un logiciel. Les activités correctives concernent celles qui corrigent les erreurs insérées lors du développement du logiciel ou lors des activités perfectives ou adaptatives de la maintenance. Les activités perfectives consistent à ajouter de nouvelles fonctionnalités ou à améliorer celles qui existent déjà. Les activités adaptatives, quant à elles, visent à adapter le logiciel à un nouvel environnement.

Il existe aussi une autre forme de maintenance; la maintenance prédictive ou préventive qui consiste à prévoir certains changements à travers le temps. Ce type d'activités peut avoir lieu à différents moments du cycle de vie du logiciel [12] [15]. Le standard IEEE 14764 de 2006 relate également ce type de maintenance : « La modification d'un produit logiciel [...] pour détecter et corriger les erreurs latentes [...] avant qu'elles deviennent des erreurs opérationnelles ». Dans ce type

d'activités, certains aspects sont considérés en particulier, l'importance de la documentation [10][17], les choix liés à l'implémentation pour augmenter la maintenabilité d'un logiciel et les activités de test. Ce type de maintenance permet de découvrir très tôt certaines erreurs réduisant ainsi leurs coûts de correction sachant que plus on tarde à découvrir une erreur plus son coût de correction risque d'être élevé. Par ailleurs, dans certains projets d'envergure, le nombre d'erreurs reportées risque d'être trop élevé par rapport à la capacité du personnel à les éliminer. Dans ce contexte, certaines erreurs vont prendre beaucoup de temps et d'autres risquent de ne pas être corrigées [11]. L'importance de faire de la maintenance préventive devient dans ces cas très important. Elle est nécessaire pour prédire les changements possibles, les composants qui risquent d'être affectés par ces changements et enfin les coûts relatifs à ces changements.

Par ailleurs, les experts s'accordent à dire que deux des activités les plus importantes de la maintenance sont la compréhension du logiciel et l'évaluation de l'effet d'un changement, c'est-à-dire l'analyse de l'impact [20]. Comme la taille des logiciels est souvent grande, il est difficile, voire impossible, pour un programmeur à la maintenance de connaître le fonctionnement entier d'un logiciel. De ce fait, lorsqu'une modification est apportée au logiciel, il est d'autant plus difficile pour le programmeur de connaître l'ampleur des répercussions liées à la modification. Par ailleurs, selon[10][15], plus on modifie un logiciel lors de la maintenance, plus on décroît ses performances et plus il risque de contenir d'autres erreurs affectant ainsi grandement sa qualité.

Pour toutes ces raisons, la maintenance doit être prise en compte lors du développement (dans le but de la faciliter) et supportée (technique, outils. etc.) lors de sa mise en œuvre.

2.3 Analyse de l'impact

L'impact du changement et son analyse représentent une des activités les plus importantes de la maintenance. Le domaine, qui est assez complexe, est souvent discuté dans la littérature, et étudié le plus souvent de façon superficielle. C'est rarement le sujet principal d'une étude à cause de la diversité des approches possibles

et de la difficulté à aller en profondeur dans ce type de problématique. L'analyse de l'impact du changement est en effet, un domaine très complexe qui dépend du paradigme et du langage utilisé.

Plusieurs types d'approches ont été proposés à travers les années par une multitude de chercheurs. La sous-section « Les approches » présente diverses approches proposées dans la littérature pour l'étude et l'analyse de l'impact du changement.

L'analyse de l'impact du changement – souvent abrégée sous le terme « d'analyse d'impact » – porte sur l'étude du comportement d'un logiciel suite à un changement. Elle consiste en un ensemble de techniques permettant d'évaluer les effets des changements réalisés sur le code source d'un logiciel [29][30]. L'analyse d'impact cherche à comprendre et à modéliser le comportement d'un changement afin de soutenir diverses tâches de la maintenance telles que la phase d'évaluation des coûts d'une modification, la phase de modification et la phase des tests de régression [27][28].

Une des techniques utilisées pour comprendre pleinement le comportement d'un changement est l'analyse de la propagation de l'impact. Il s'agit d'un processus itératif d'analyse des impacts des changements utilisé pour assurer la consistance et l'intégrité d'un système avant et après que les changements aient été implémentés [31][32][33].

Certaines approches proposées, comme celles présentées dans [7] et [6], visent une seule phase de la maintenance. Dans [7], les auteurs s'intéressent à l'analyse d'impact pour pouvoir mieux évaluer les coûts liés à un changement donné. Dans [6], les auteurs s'intéressent à l'analyse d'impact pour mieux orienter le processus de test de régression en identifiant les parties impactées par les changements et qui doivent être testées à nouveau. D'autres approches, comme celles proposées dans [1] et [8] et celle qui est présentée dans ce mémoire, ne visent pas une phase particulière. Elles sont plus générales. Elles présentent, aussi, un modèle d'analyse d'impact.

Pour supporter l'analyse de l'impact, ces approches se basent ou utilisent différentes techniques. L'approche proposée dans [7] est basée sur les métriques. Celle présentée

dans [6] utilise les deux versions consécutives du programme, les tests ainsi que les graphes d'appels. Les approches proposées dans [1] et [8] utilisent le modèle de classes du logiciel. Implicitement, la source des éléments analysés par les approches ainsi que leur façon de faire (démarche) les classe en deux groupes d'analyses : les analyses prédictives et les post-analyses. La distinction n'est que rarement explicitement effectuée. Cette distinction sera abordée dans ce qui suit, car elle est importante pour la bonne compréhension de notre modèle.

2.3.1 Post-analyse

L'analyse de l'impact d'un changement peut s'effectuer avant ou après que celui-ci ait eu lieu. Lorsqu'on fait de la post-analyse, cela signifie que le changement a eu lieu et qu'on se retrouve en possession de deux versions d'un même élément d'analyse (code, modèle de classes, graphe d'appels, etc.). On pourra alors analyser les impacts ayant eu lieu pour mieux supporter le processus de maintenance. Comme le changement a été effectué, ce type d'analyse sera généralement utile pour les phases tardives de la maintenance en l'occurrence la phase des tests de régression.

2.3.2 Analyse prédictive

Lorsqu'on fait de l'analyse prédictive, cela signifie que le changement n'a pas eu lieu et qu'on se retrouve en possession d'une seule version d'un même élément d'analyse (code, modèle de classes, graphe d'appels, etc.). Ces approches sont plus proactives que les premières. Dans ce cas, on cherchera effectivement à prédire l'impact pour mieux supporter les différentes activités du processus de maintenance, évaluation des coûts liés à un changement, évaluation possible de plusieurs alternatives ou solutions, etc. Il est clair que ces approches permettent d'atteindre plus d'objectifs que les premières. Elles sont toutefois plus complexes à élaborer et à mettre en œuvre.

2.4 Les approches

A. April et A. Abran ont réalisés des travaux [38][39] sur l'évaluation du processus de maintenance. Ils soutiennent qu'il existe beaucoup de modèles portant sur l'évaluation de la qualité du développement du logiciel, contrairement à l'évaluation

de la qualité du processus de maintenance; presque inexistant. Dans [38], les auteurs font une première analyse des modèles existants et présentent les bases de leur propre modèle. Ils souhaitent ainsi développer un outil supportant le processus de maintenance et permettre son amélioration.

M.K. Abdi et al. [7] ont proposé une technique probabiliste avec les réseaux bayésiens pour vérifier des résultats obtenus précédemment. Ces réseaux permettent de faire la corrélation entre certaines métriques de couplage et de complexité et la capacité de changeabilité d'un système orienté objet. Avec ces métriques, ils peuvent obtenir une indication sur le « danger » d'effectuer un changement dans un système. Le réseau bayésien utilisé dans ce travail leur permet de vérifier ces corrélations. Leurs expériences montrent qu'une de leurs hypothèses est en contradiction avec l'une des métriques retenues. La métrique en question est liée à la taille du logiciel qui n'est en quelque sorte pas prise en considération dans le réseau.

Le même groupe de chercheurs a utilisé dans [1] et [8] les travaux de A. Chaumun et H. Kabaili qui seront présentés par la suite. Ces derniers ont proposé un modèle d'impact basé sur le modèle de classes des programmes. M.K. Abdi et al. utilisent ce modèle dans le but d'évaluer la corrélation entre les métriques de couplage et l'impact du changement.

Dans [6], ce groupe de chercheurs a implémenté une technique d'analyse d'impact dont l'objectif est de supporter les tests de régression. Ils ont mis au point un *plug-in,* nommé Chianti fonctionnant dans l'environnement Eclipse, qui analyse diverses versions d'un même programme en langage Java via l'environnement CVS. Dans leur projet, ils font donc de la post-analyse d'impact. Ils ne cherchent pas à prédire l'impact d'un changement, mais ils veulent plutôt identifier les différents impacts suite à une modification effectuée, en analysant deux versions consécutives d'un même programme.

Leur technique se déroule en quatre grandes étapes. Premièrement, un ensemble A de changements entre deux versions d'un programme est extrait. Ensuite, un graphe d'appels est construit pour chaque test du programme (ensemble T). Puis, un

ensemble T' de tests ayant été affectés par les changements de l'ensemble A est déterminé. Enfin, la dernière étape consiste à déterminer l'ensemble des changements de l'ensemble A ayant affecté chacun des tests de l'ensemble T'.

Tableau I

Changements unitaires considérés dans Chianti (tiré de [6]).

AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of a method
LC	Change virtual method lookup
AF	Add a field
DF	Delete a field
CFI	Change definition of a instance field initializer
CSFI	Change definition of a static field initializer
AI	Add an empty instance initializer
DI	Delete an empty instance initializer
CI	Change definition of an instance initializer
ASI	Add an empty static initializer
DSI	Delete an empty static initializer
CSI	Change definition of an static initializer

L'ensemble A des changements est un ensemble de changements unitaires qui sont pour la plupart de niveau méthode. Cet ensemble comprend les changements de base notamment, l'ajout d'une méthode (**AM**), le retrait d'une méthode (**DM**), le changement du corps d'une méthode (**CM**), l'ajout d'un champ (d'un paramètre) à la méthode (**AF**), le retrait d'un champ à la méthode (**DF**), l'ajout d'une classe (**AC**), le retrait d'une classe (**DC**) et le changement « lookup » (**LC**). L'ensemble complet des changements utilisés avec Chianti est donné dans le tableau I.

Lors de la première étape du processus d'analyse, chacun de ces changements est identifié entre les deux versions du même programme. Cela permet de sélectionner l'ensemble T' des tests ayant été affectés par ces changements. Pour chacun de ces tests, on identifie l'ensemble des changements ayant affecté un test donné car si celui-ci échoue on saura la cause de cet échec. Un outil supportant la mise au point est disponible à cet effet. Pour déterminer cet ensemble de changements affectant un test donné, un ensemble de règles d'ordonnancement des changements a été établi sur un principe intuitif. On part du principe que tous les changements introduits entre

deux versions d'un programme peuvent être effectués dans un ordre où le code source restera toujours syntaxiquement correct. Par exemple, avant de retirer une classe (**DC**), on devra retirer ses méthodes (**DM**). **DM** est un pré requis à **DC**, donc, si **DC** a affecté un test, les **DM** aussi. Il s'agit d'un exemple simple, mais les règles établies respectent ce type d'ordonnancement.

Plusieurs conclusions ont été établies: plus l'écart entre deux versions est grand (en temps ou en versions), plus le nombre de changements affectant chacun des tests augmente. À cela, on pourrait ajouter, que plus le nombre de changements affectant un test augmente, plus il est exigeant pour un programmeur de retrouver la ou les sources d'erreurs introduites à l'origine de l'échec d'un test.

Malheureusement, leurs travaux n'ont pas focalisé sur les performances de leur modèle. Il est donc impossible de savoir si leur modèle donne des résultats précis ou non quant aux changements qui ont affecté un test. La précision de ce type de modèle, dans ce contexte, est un facteur très important.

Chaumun [2][4][5][9][29] a pour sa part développé une approche pour l'analyse d'impact basée sur le diagramme de classes et le code source des programmes (langage C++). Un ensemble de changements unitaires pouvant être effectués dans le code et affectant la structure des classes a été établi. Ces changements se situent soit au niveau de la classe (ajout d'une classe, suppression d'une classe, etc.), soit au niveau des méthodes (ajout d'une méthode, suppression d'une méthode, etc.) ou encore au niveau d'un attribut de classe (ajout d'un attribut, suppression d'un attribut, etc.). Une liste de 19 changements de niveau classe, 35 changements de niveau méthode et 12 changements de niveau attribut pour un total de 66 changements a été établie.

Pour chacun de ces changements, une règle d'impact est définie selon les relations entre classes. Les relations possibles sont l'agrégation (**G**), l'association (**S**), l'invocation (**I**), l'héritage (**H**), la relation spécifique au langage C++ qui est le « *friendship* » (**F**) et l'autoréférence locale (**L**) lorsque l'impact se retrouve à l'intérieur même de la classe où a lieu le changement.

Par exemple, si on prend la règle d'impact « changement de type d'un attribut de classe », soit « S + L », cela signifie que toutes les classes associées à la classe où est situé le changement de type de l'attribut et la classe elle-même seront des classes impactées. Le tableau complet des règles d'impacts du modèle de Chaumun pour le langage C++ est disponible à l'annexe A.

Kabaili [2][3][4][5] a poursuivi les recherches de Chaumun en adaptant le modèle pour le langage Java. Il en résulte 15 changements de niveau classe, 25 changements de niveau méthode et 12 changements de niveau attribut pour un total de 52 changements. Les 14 changements unitaires qui ont été retirés du modèle de base (C++) sont les changements liés à la notion de classe virtuelle. Dans ce langage, il y a des classes non-virtuelles, virtuelles et purement virtuelles. Les classes non-virtuelles correspondent aux classes non-abstraites en Java et les classes purement virtuelles aux classes abstraites. Comme il n'y a pas d'équivalence entre les classes virtuelles en C++ et en Java, tous les changements qui leur sont liés dans le modèle d'impact pour C++ disparaissent dans le modèle d'impact pour Java. Les règles d'impact dans le modèle pour Java restent les mêmes à l'exception de la règle relative à la relation « friendship » (**F**) qui est retirée puisque inexistante en Java. Le tableau des règles d'impacts du modèle de Chaumun pour le langage Java est disponible a l'annexe A.

Dans une autre étude [3], Kabaili discute de l'utilisation du modèle d'analyse d'impact de Chaumun qu'elle a adapté pour le langage Java pour faire de l'analyse d'impact en cascade (« *ripple effect* »).

L'analyse d'impact en cascade est l'analyse des impacts avec un effet de propagation (plusieurs niveaux d'impact : directs et indirects). Lorsqu'on effectue un changement, cela forcera d'autres changements. Ces sous changements forceront eux aussi d'autres changements et ainsi de suite. L'analyse d'impact en cascade cherche donc à identifier cette propagation.

Dans [3], Kabaili suggère d'utiliser le modèle de Chaumun en deux étapes. La première étape consiste à utiliser le modèle tel que prévu sur le changement initial afin d'identifier l'ensemble des classes impactées. La deuxième étape demande de

tenter d'identifier la nature des impacts qui auront lieu dans les classes impactées. Une fois ces deux étapes accomplies, on les exécute de nouveau, mais cette fois-ci par rapport aux impacts dont la nature a pu être identifiée, en les considérant comme des changements initiaux. Dans le contexte présenté dans [3], on possède les deux versions successives d'un même logiciel. Il est donc possible de connaître les changements qui ont eu lieu. C'est la seule utilisation en post-analyse du modèle de Chaumun qui est présentée dans la littérature.

Comme le modèle de Chaumun, adapté pour le langage Java par Kabaili, est le modèle le plus complet et le plus similaire à celui présenté dans ce mémoire, il servira de référence. De plus, une section du chapitre 3 lui sera consacrée pour davantage de précision.

D. St-Yves [19] a travaillé sur une approche d'impact basée sur les graphes de contrôle réduits aux appels. Il soulève les lacunes, en termes de précision, dues à l'utilisation d'une méthode d'analyse uniquement basée sur les graphes d'appels directs, qui fournissent peu d'indices sur le comportement d'un logiciel (structures des appels et leur contrôle). On rappellera qu'un graphe d'appels direct ne fournit que la liste des méthodes M_i appelées par une méthode M. C'est pour cette raison que D. St-Yves utilise un graphe de contrôle réduit aux appels, c'est-à-dire un graphe lui permettant d'obtenir les séquences d'appels des méthodes (un modèle plus expressif que les graphes d'appels).

À la figure 2, on a en (3.1) le code source d'une méthode, en (3.2) le graphe d'appels direct correspondant et en (3.3) son graphe de contrôle réduit aux appels.

La technique proposée par D. St-Yves et al. [21][22][23] utilise les graphes de contrôle réduit aux appels. Il s'agit d'une forme réduite des graphes de contrôle classiques. Ils représentent un modèle intermédiaire entre les graphes de contrôle traditionnels et les graphes d'appels directs. Dans son implémentation – un « plug-in » dans l'environnement de développement Éclipse – l'utilisateur sélectionne dans une méthode la partie de code qu'il projette de modifier. À l'aide des graphes de

contrôles réduits aux appels, une analyse d'impact prédictive sera effectuée pour déterminer quelles parties du logiciel seront impactées (impacts directs et indirects).

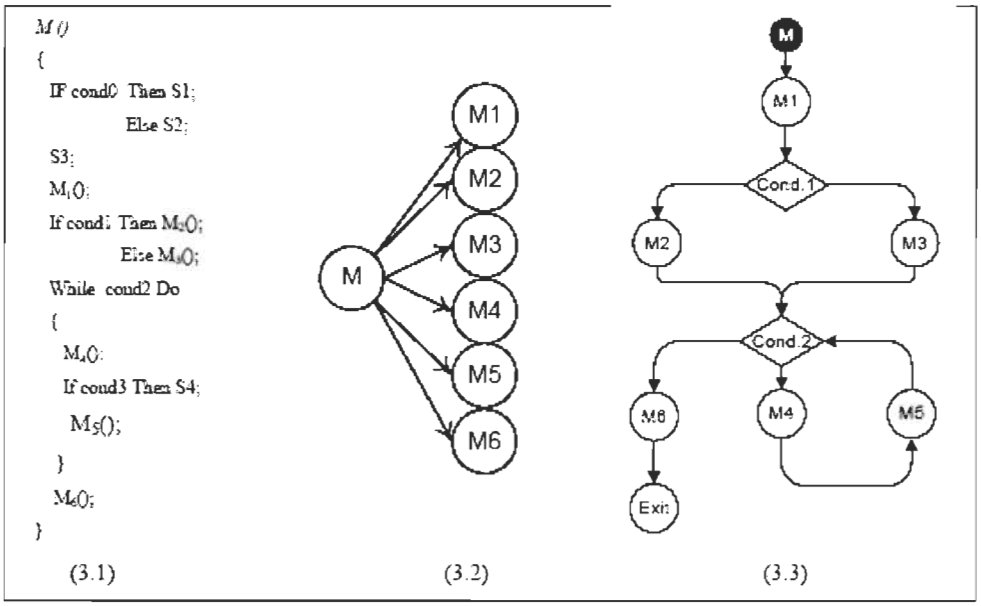

Figure 2 Code source d'une méthode, graphe d'appels et graphe de contrôle réduit aux appels correspondants [19].

La technique considère comme méthodes potentiellement impactées, suite au changement d'une méthode M, toutes les méthodes appelant M, et appelées après M. Le principe a déjà été utilisé dans [24] pour les graphes d'appels. L'utilisation des graphes de contrôle réduits aux appels est importante dans cette technique puisqu'ils apportent plus de précision permettant ainsi de réduire l'impact contrairement aux graphes d'appels.

CHAPITRE 3

PRÉSENTATION DU MODÈLE D'ANALYSE D'IMPACT

Dans le cadre de ce mémoire, un Modèle d'Impact du Changement pour le langage Java (MICJ) a été développé dans le but d'offrir un modèle permettant une analyse systématique et précise de l'impact du changement pour les applications Java.

Ce chapitre présente en premier les détails du modèle d'analyse d'impact présenté par Chaumun et Kabaili ([1][3][4][5][9]). Le fonctionnement et l'utilisation de MICJ sont présentés par la suite. Le modèle de Kabaili sert de référence dans le chapitre suivant pour mieux permettre l'évaluation du MICJ.

3.1 Modèle de Chaumun et Kabaili

Tel que mentionné dans le chapitre précédant, le modèle d'impact de changement pour le langage Java présenté par Kabaili se base sur celui de Chaumun pour le langage C++ . Dans la suite de ce chapitre, et au cours des chapitres suivants, nous parlerons toujours du modèle de Chaumun en faisant référence à son adaptation pour le langage Java. Il s'agit du modèle le plus élaboré dans la littérature. Il servira donc de référence dans les expérimentations qui seront présentées ultérieurement, même si celles-ci focalisent sur les résultats du modèle MICJ et non pas sur ceux fournis par le modèle de Chaumun.

Le modèle de Chaumun fait mention de 52 changements unitaires possibles dans du code Java. Un changement unitaire dans un modèle est un changement ne pouvant pas être décomposé en plusieurs changements. Par exemple, soit la ligne de code suivante :

```
public static void main ( String args[] )
```

Si on change cette ligne de code par :

```
private void main ( int args[] )
```

Le changement de « l'écriture de la méthode » ne serait pas un changement unitaire car elle est composée de plusieurs changements unitaires selon le modèle de Chaumun, soit : changement de visibilité de la méthode, passage de statique à non-statique de la méthode et changement de la signature de la méthode (correspond au changement de type ou du nombre de paramètres d'une méthode).

Le modèle de Chaumun divise ces changements en trois catégories selon le niveau auquel ils s'appliquent : les changements de niveau classe, les changements de niveau méthode et les changements de niveau attribut.

Chaque changement du modèle de Chaumun a une règle d'impact qui détermine quelles classes seront impactées selon leurs relations avec la classe où le changement a eu lieu. Les relations considérées par le modèle de Chaumun sont l'association (A), l'agrégation (G), l'héritage (H) et la classe elle-même où le changement a eu lieu, la relation est locale (L) dans ce cas. De plus, il y a le lien d'invocation (I) et certaines règles font exceptionnellement référence à un autre changement du modèle. Par ailleurs, il est possible pour une classe *N* d'avoir l'ensemble des relations vis-à-vis d'une classe *M*.

Pour le modèle de Chaumun, l'impact de changement est l'ensemble des classes impactées par un changement unitaire donné. Le modèle de Chaumun est un modèle d'impact de type prédictif de niveau classe, c'est-à-dire que son niveau de prédiction de l'impact a pour limite de précision l'identification des classes impactées. Selon [4], le modèle de Chaumun est sensé prédire les impacts certains et incertains. Cependant, il ne fait pas la distinction entre les deux lors de l'analyse. Des précisions seront apportées sur la notion de certitude de l'impact ultérieurement dans ce chapitre.

le tableau II présente quelques exemples de règles du modèle de Chaumun.

Tableau II

Exemples de règles d'impact du modèle de Chaumun.

Identifiant du changement	**Nom du changement**	**Règle d'impact**
c.3.1.2	Changement de la classe : Abstraite → non-abstraite	H+L
v.1.5.1	Changement de visibilité de la variable : publique → privée	S~H
v.1.5.2	Changement de visibilité de la variable : Protégée → privée	SH

Ainsi, pour la règle *c.3.1.2*, lorsqu'une classe abstraite devient non-abstraite, son impact se retrouve dans les classes qui héritent localement. Le « + » à la signification d'un « ou » logique. Le « ~ » (parfois le « ' » est utilisé aussi) permet d'indiquer la négation. Donc, pour le changement *v.1.5.1*, il y aura un impact dans les classes en association qui n'héritent pas de la classe. La combinaison de deux relations telle que présentée avec la règle *v.1.5.2* avec le « + » est la représentation du « et » logique. Par conséquent, il y a impact dans les classes qui ont une relation d'association et qui héritent de la classe changée.

Dans son mémoire [9], Chaumun explique comment il a obtenu ces règles d'impacts pour chaque changement. Comme le mémoire discutait du modèle pour C++, aux liens précédents s'ajoutent le lien « friendship » (F) qui est une spécificité du langage. Une classe *A* peut posséder un lien de chaque sorte (A, G, H, I ou F) avec une classe *B*, et pour chaque combinaison de lien il y a ou non impact sur la classe *A* avec le changement dans la classe *B*. Si on reprend l'exemple de l'annexe A de [9] pour la règle d'impact du retrait d'une méthode, on obtient l'expression canonique « SGHIF' + SG'HIF' + S'GHIF' + S'G'HIF' » qui a été réduite à « I+ie(3.1.1) ». L'exemple complet, nommé *calcul d'impact de Chaumun*, avec le tableau et les explications sur la règle d'impact tiré du mémoire de Chaumun est donné dans «Calcul de l'impact de Chaumun» à l'annexe A.

Le modèle de Chaumun a pour principale lacune son niveau d'analyse d'impact qui est à granularité forte et abstrait (niveau classe). Cet aspect a cependant l'avantage de procurer des règles d'impacts simples et de disposer d'un modèle facile d'utilisation. Néanmoins, le modèle, tel que présenté, impacte certaines classes sans prendre en compte la nature du changement effectué, ni les conséquences du changement en termes d'impact. Il n'y a, donc, aucune spécification à propos des classes qui seront impactées en dehors de celles reliées à la classe où a lieu le changement. Par ailleurs sans la spécification précise de la nature de l'impact, il est impossible d'effectuer une analyse en cascade de l'impact d'un changement en utilisant uniquement le modèle de Chaumun. À cause de ce niveau d'impact élevé (abstraction), le modèle est moins précis et moins efficace. En effet, l'ajout ou le retrait d'une relation dans une règle d'impact modifie grandement la signification de cette dernière. Par exemple, pour un changement *x*, si on avait la règle d'impact « L », puis qu'on découvre une exception faisant en sorte que certaines classes associées pourraient être impactées en remplissant des conditions exceptionnelles et qu'on change conséquemment la règle d'impact pour « L+A », alors toutes les classes associées seront considérées comme impactées même si elles ne remplissent pas les conditions exceptionnelles. Donc, avoir la règle « L » n'est pas précis, et « L+A » non plus.

3.2 Le modèle MICJ

3.2.1 Objectifs

Le modèle MICJ se veut un modèle d'analyse de l'impact du changement pour le langage Java. Ce modèle fournit une quantité suffisante d'information sur l'impact lié à un changement permettant ainsi l'analyse d'impact en cascade avec précision. De plus, le modèle est suffisamment souple afin que ses règles d'impact puissent être modifiées pour améliorer ses performances.

Le modèle MICJ est un modèle supportant la post-analyse et l'analyse prédictive. Nous avons délibérément choisi de nous concentrer sur ses capacités d'analyse prédictive dans ce mémoire.

Par ailleurs, pour évaluer la qualité du modèle, nous avons utilisé deux mesures (largement utilisées dans la littérature pour ce type d'évaluation [25]) dans les expérimentations effectuées : la précision et le rappel. La précision est définie comme la capacité du modèle à prédire que des impacts auront lieu, soit le nombre d'impacts prédits qui sont effectivement survenus sur le nombre d'impacts prédits au total par le modèle. Le rappel est quant à lui la capacité du modèle à prédire tous les impacts réels, soit le nombre d'impacts que le modèle prédit correctement sur le nombre d'impacts réels au total.

Le modèle MICJ a pour objectifs d'avoir un niveau intéressant de précision et de rappel. Il apporte un niveau de détail dans l'analyse de l'impact en offrant à la fois la possibilité de faire de l'analyse d'impact en cascade et un maximum d'informations pour diverses éventualités d'utilisation. Ces éventualités pourraient, entre autres, concerner l'optimisation des tests de régression, le support dans l'estimation des coûts liés à l'implantation d'un changement, l'aide à la mise en place d'un changement, etc.

3.2.2 Origine du modèle

Comme mentionné précédemment, la notion d'impact en cascade correspond à la propagation des impacts dans un programme. Rares, en fait, sont les approches qui permettent effectivement cette analyse en cascade. Dans [6], les auteurs génèrent un ordonnancement logique des changements pour faire en sorte qu'à la suite de chaque changement, le programme soit toujours compilable. Cela permet uniquement, à partir d'un modèle réduit des changements unitaires, une forme d'analyse en cascade déduite en post-analyse.

Avec le modèle de Chaumun, on obtient une analyse uniquement prédictive de l'impact et de niveau classe. Le modèle est par contre relativement complet au niveau des changements unitaires. Il ne permet toutefois pas une analyse en cascade automatisable tel que discuté précédemment dans la revue de la littérature.

Le modèle MICJ, pour atteindre ses objectifs, s'est inspiré du modèle de Chaumun. Un ensemble de changements unitaires ont été identifiés qui, tout comme dans le

modèle de Chaumun, sont soit de niveau classe, soit de niveau méthode ou bien de niveau attribut. Cependant, pour plus de précision, le modèle MICJ ne spécifie pas l'impact pour chaque changement comme un ensemble de classes liées à celle où le changement a eu lieu. Il spécifie pour chaque changement un ensemble d'impacts de changements en précisant l'emplacement de ces impacts apportant ainsi plus de précision pour une meilleure analyse de l'impact en cascade contrairement au modèle de Chaumun.

3.2.3 Généralité du modèle

3.2.3.1 Changements structurels et non-structurels

Pour apporter une grande précision au modèle MICJ, beaucoup de notions ont été utilisées. Le modèle inclut un grand nombre de changements atomiques. Un changement atomique est la plus petite unité de changement. Il ne peut pas être décomposé en plusieurs changements atomiques. Par exemple, le changement de signature d'une méthode ne serait pas un changement atomique dans le modèle MICJ parce qu'il pourrait être décrit par plusieurs autres changements atomiques du modèle tels que l'ajout d'un paramètre, changement de type de retour et changement de visibilité de la méthode.

Ces changements atomiques se divisent en deux groupes distincts: les changements structurels et les changements non-structurels. Un changement structurel est un changement qui affecte la structure de la classe et qui est visible sur un diagramme de classes complet. À titre d'exemple, citons l'ajout ou le retrait d'une méthode. Un changement non-structurel est un changement qui n'affecte pas la structure de la classe et qui n'est pas visible sur un diagramme de classes complet. Les changements non-structurels sont possibles dans le corps des méthodes. Par exemple, l'ajout ou le retrait d'un appel d'une méthode.

Le modèle MICJ possède 48 changements structurels. Ils sont divisés en trois niveaux : changement de niveau classe (ajout d'une classe, suppression d'une classe, etc.), changement de niveau méthode (ajout d'un paramètre, changement de type de retour, etc.) et changement de niveau attribut (ajout d'un attribut, changement de

visibilité de l'attribut, etc.). Il y a 9 changements atomiques de niveau classe, 24 de niveau méthode et 15 de niveau attribut. La liste des changements et de leur abréviation est disponible à l'annexe B dans le tableau des changements atomiques.

Les changements non-structurels sont au nombre de 26. La liste des changements non-structurels et leur abréviation est disponible à l'annexe B dans le tableau des changements non-structurels.

Il existe pour les changements non-structurels des exceptions (quatre en tout). Ce sont des changements pour lesquels il y a des sous changements. Il y a « l'ajout d'appel de méthode » qui est décomposé en changement atomique « ajout d'appel de méthode héritée » ou « ajout d'appel de méthode statique », son contraire « retrait d'appel de méthode » décomposé en « retrait d'appel de méthode héritée » ou « retrait d'appel de méthode statique », « ajout d'appel d'attribut » qui peut être plus spécifiquement un « ajout d'appel d'attribut hérité » et son contraire « retrait d'appel d'attribut » un « retrait d'appel d'attribut hérité ». Ces quatre changements sont plus généraux que les autres. Leurs sous-changements sont des spécificités d'eux-mêmes et permettent de préciser certaines règles d'impacts. Par exemple, « ajout d'appel de méthode héritée » est un cas spécifique de « ajout d'appel de méthode ». Cela veut également dire que si on recherche comme impact « ajout d'appel de méthode », on recherche du même coup « ajout d'appel de méthode héritée ». Les deux derniers changements sont les changements « ajout d'utilisation de variable » et « retrait d'utilisation de variable » qui sont aussi des cas généraux. Cela permet de spécifier que partout où une variable est utilisée, il y aura dans un cas ajout d'utilisation et dans l'autre retrait d'utilisation de cette variable peu importe le type d'utilisation : déclaration, initialisation, utilisation comme paramètre dans un appel, utilisation dans une opération (addition, soustraction, etc.) et toutes autres formes d'utilisation.

3.2.3.2 Notion de certitude

Le modèle MICJ apporte une notion de certitude qui n'existe pas dans le modèle de Chaumun à cause de ses règles d'impact trop générales. Il est important de bien saisir cette notion, car elle permet de mitiger l'analyse de l'information que procure l'analyse d'impact avec le modèle MICJ. Pour cela, nous allons utiliser deux

changements avec uniquement un des éléments d'impact de leur règle d'impact pour rendre explicite la signification de la certitude.

Prenons comme premier exemple le changement atomique « retrait d'un attribut de classe ». Cela aura comme impact de retirer toutes les utilisations de cet attribut. Pour pouvoir compiler le code, en retirant l'attribut, on devra aussi retirer ses utilisations. On parle donc ici de certitude ou d'impact certain.

Prenons comme second exemple le changement atomique « ajout d'un attribut de classe ». Normalement, si on ajoute un attribut à une classe, c'est qu'on a l'intention de l'utiliser. Sinon, ce serait une modification inutile, mais reste toutefois possible. Donc, on peut prédire qu'il y aura comme impact l'ajout d'utilisation de l'attribut. Par contre, contrairement au cas du retrait de l'attribut, on n'est pas certain de retrouver l'impact car le code continuera d'être compilable même si on ne fait pas l'ajout d'utilisation de l'attribut. On parle donc ici d'un impact incertain. Le changement permet cet impact (cet autre changement), mais ne l'exige pas.

Cela fait en sorte que l'on peut être certain qu'un ensemble d'impacts aura lieu et incertain qu'un autre ensemble d'impacts aura lieu, mais qui reste possible.

Le modèle MICJ, dans ses règles d'impact, fait la distinction entre les impacts certains et incertains pour donner une information supplémentaire à l'analyste en précisant ce qui est certain et qu'est-ce qui sera impacté de ce qui pourrait être éventuellement impacté.

Le modèle de Chaumun ne fait pas la distinction dans ses règles d'impact. Il est, cependant, censé considérer les deux impacts certains et incertains.

3.2.3.3 Relations entre classes

Pour le modèle MICJ, les relations entre les classes seront utilisées afin de définir les règles d'impact de changement. Les relations seront utilisées pour préciser chaque impact et indiquer dans quelles classes on doit chercher l'impact.

```
public class M {}

public class N
{
   M unAttribut;

   public N (M unAttribut)
   {
      this.unAttribut = unAttribut;
   }

   public M methodeA()
   {
      return new M();
   }

   public void methodeB()
   {
      new M();
   }
}
```

Figure 3 Relation d'association.

Le modèle MICJ considère quatre relations entre classes possibles : l'association *A*, l'héritage *H* (hérite de), l'héritage descendant-ascendant $H^{\wedge}$ (en cas d'impact du descendant sur l'ancêtre) et la pseudo-relation local *L* (la classe elle-même). Les relations sont toujours considérées par rapport à la classe M où le changement a lieu. Cela permet d'avoir les classes N_i ayant une relation avec la classe M.

Une relation d'association *A* de la classe N à M est considérée si la classe N possède un attribut de type M, si une méthode de la classe N reçoit en paramètre un objet de type M, si une méthode de la classe N retourne un objet de type M ou si une méthode de la classe N déclare ou initialise un objet de type M. Voir l'exemple avec les cas mentionnés en gras à la figure 3.

On considère une relation d'héritage *H* de la classe N à M si la classe N hérite (ou étend) de la classe M. On retrouvera concrètement dans le code à la signature de la classe « extends M ». Voir l'exemple avec le cas mentionné en gras à la figure 4.

```
public class M {}

public class N extends M
{}
```

Figure 4 Relation d'héritage.

Une relation d'héritage descendant-ascendant H^S de la classe N à M est considérée si la classe M hérite (ou étend) de la classe N. On retrouvera concrètement dans le code à la signature de la classe « extends N ». Chaque classe M_i ne peut avoir qu'une seule classe N_i ayant la relation d'héritage ascendant à cause de la règle d'héritage simple du langage Java. Voir l'exemple avec le cas mentionné en gras à la figure 5.

```
public class N {}

public class M extends N
{}
```

Figure 5 Relation d'héritage descendant-ascendant.

On considère une pseudo-relation locale L de la classe N à M si la classe N et M correspondent à la même classe. Donc, chaque classe N_i a une pseudo-relation locale par rapport à elle-même.

Le modèle MICJ permet l'existence de plus d'une relation entre une classe N et M. Prenons l'exemple à la figure 6.

```
public class N extends M
{
      public M m;
}
```

Figure 6 Relations mutliples entre classes.

La classe M a alors la classe N comme classe associée (A) et comme classe qui hérite (H).

3.2.3.4 Cibles

Les cibles ont la même fonction que les relations entre classes : spécifier les endroits potentiels d'un impact. Par contre, celles-ci donnent une précision supérieure à celle des relations entre classes et peuvent être utilisées en tant que condition sur la certitude de l'impact. La liste des cibles est disponible dans « Tableau des cibles » de l'annexe B.

3.2.4 Règle d'impact

Le modèle MICJ possède plusieurs ensembles d'éléments distincts permettant d'écrire ses règles d'impacts : les changements atomiques, les changements structurels, les changements non-structurels, les règles d'impacts, les éléments de changement, les éléments d'obligation, les éléments de précisions, les conditions et les cibles. Il est donc important de comprendre la signification de chacun de ces éléments.

Le premier ensemble à connaître est l'ensemble des changements atomiques *CA*. Cet ensemble est composé de deux autres ensembles complémentaires : les changements structurels *CS* et les changements non-structurels *CNS*.

Une règle d'impact *RI* est donnée pour chacun des ensembles. La règle d'impact est définie par des éléments de changement *EC* qui sont les éléments qui définissent comment et où auront lieu les impacts. Une règle d'impact permet d'obtenir un ensemble d'impact *I* de changements atomiques appartenant à *CA* qui sont les changements que devront ou pourraient être effectués dans le code.

Un *EC* est composé d'un ou plusieurs changements atomiques appartenant à *CA* liés par un « OU » ou un « ET ». Le « OU » désigne qu'au moins l'un des deux changements aura lieu. Le « ET » stipule que les deux changements auront lieu. Un *EC* est aussi composé d'un ou plusieurs indicateurs de relation *IR* et, possiblement, d'une précision *Pr* et d'une condition *Co*. Un *EC* possède aussi un élément d'obligation *EO*.

Un *IR* est une relation existant entre classes et permettant de déterminer dans quelles classes peut se retrouver le *EC*.

Un élément de précision *Pr* est un élément *e* appartenant à l'ensemble des cibles *Ci*. Un élément de précision permet de préciser où il peut y avoir impact parmi les classes correspondantes aux *IR* du *EC*. Cela permet de savoir où chercher le ou les changements atomiques du *EC* parmi et à l'intérieur des classes.

Une condition *Co* est un élément *e* appartenant à l'ensemble des cibles *Ci*. Elle s'applique sur le *EO* du *EC*. Si le *Co* est respecté, alors le *EO* est considéré tel que défini dans le *EC*. Dans le cas contraire, le *EO* est inversé.

Un élément d'obligation *EO* définit la certitude d'un ou plusieurs changements atomiques du *EC* dont on peut être certain qu'ils arriveront ou pas.

La liste des différents éléments – changements atomiques et cibles – sont disponibles dans les « tableau des changements structurels », « tableau des changements non-structurels » et dans le « tableau des cibles » de l'annexe B.

Voici un exemple de règle d'impact :

$\mathbf{Ct_{an}} \rightarrow Mt_{an}$ {Rma} (L) + [Nai (L, A, H)]

Cette règle à pour signification que le passage d'une classe abstraite à non-abstraite (Ct_{an}) a comme impact le passage des méthodes de abstraite à non-abstraite (Mt_{an}) dans cette classe (*L*) pour les méthodes abstraites (*Rma*) et peut-être, avec incertitude (*[]*), l'ajout d'initialisation d'objet de la classe (*Nai*) localement, dans les classes associées et dans les classes qui héritent (*L, A, H*).

L'ensemble des règles d'impacts est disponible dans le « tableau des règles d'impacts » à l'annexe B.

3.2.5 Utilisation des règles d'impacts

Connaître la structure d'une règle d'impact n'est pas suffisant pour parvenir à l'utiliser correctement. Un certain nombre d'éléments sur la technique d'utilisation doit être connu.

Tout d'abord, il faut comprendre que le modèle MICJ a été conçu dans l'optique d'être utilisé aussi bien en analyse prédictive qu'en post-analyse ; le modèle contient donc l'information pour les deux types d'analyse. Par contre, certaines informations ne sont utiles que pour l'un des deux types d'analyse. La technique d'utilisation pour une analyse prédictive sera principalement présentée ici, puisque les expérimentations ont été effectuées avec ce type d'analyse.

3.2.5.1 Notion de certitude dans la post-analyse

Comme ce document focalise sur l'analyse prédictive du modèle MICJ, Il est important de préciser la différence entre l'utilisation du modèle en analyse prédictive et en post-analyse. En post-analyse, la notion de certitude disparaît et, en conséquence, la notion de condition aussi.

La notion de certitude n'existe plus car elle perd tout son sens en post-analyse : il y a eu impact ou il n'y a pas eu; il y a eu un changement dans le code ou il n'y en a pas eu. On ne peut pas être incertain que la syntaxe ou la sémantique d'une ligne de code a changé une fois qu'on a la version originale et la version modifiée. Si on retrouve l'impact tel que spécifié dans la règle, qu'il soit prédit certain ou non, c'est qu'il a eu lieu.

La notion de certitude et d'incertitude dans l'utilisation du modèle MICJ en post-analyse ne peut avoir qu'une utilité statistique, c'est-à-dire que cela peut permettre de recueillir des informations sur la probabilité qu'un impact spécifié incertain ait effectivement eu lieu. Bref, des statistiques sur l'analyse prédictive et non pas sur la post-analyse. Une statistique similaire sera d'ailleurs extraite et expliquée dans les expérimentations.

3.2.5.2 Interprétation d'une règle d'impact

Pour bien interpréter une règle d'impact, il faut, comme il a été mentionné préalablement, comprendre que le modèle est prévu pour faire de la post-analyse et de l'analyse prédictive. Il faut aussi comprendre que le modèle est spécifique au langage Java.

Le premier élément à connaître est que tout impact est interprété seulement s'il est applicable. Cela signifie qu'un certain nombre d'impacts sont détectables uniquement en post-analyse. En analyse prédictive, dans le contexte de ce travail, il y a donc un certain nombre d'impacts qui ne peuvent pas être calculés, car le modèle ne prend pas en compte les intentions du changement. Par exemple, c'est le cas du changement « ajout d'une classe », qui a comme impact « [Mpa (A)] + [Aa (A)] + [Nad (A)] », c'est-à-dire l'ajout de paramètres (de type de la classe), ajout d'attributs de classe et l'ajout de déclarations d'objet de la classe dans les classes associées. En somme, on ajoute une classe afin de pouvoir l'utiliser. Le problème, c'est qu'en analyse prédictive, on ne peut pas appliquer cette règle, car la seule prémisse que l'on possède, c'est l'ajout de la classe. On n'a aucune information sur les classes qui auront une relation avec la classe ajoutée, ce qui empêche la prédiction du moindre impact. C'est en ce sens que certains impacts – la totalité des impacts dans le cas de l'ajout d'une classe – ne peuvent pas être applicables. Il ne faut donc pas en tenir compte dans l'analyse prédictive à ce moment-là.

Un autre exemple d'impact non applicable se situe dans le changement « ajout d'un paramètre » à une méthode abstraite. Un des impacts est l'utilisation du paramètre dans la méthode ce qui signifie que la méthode dans laquelle on ajoute le paramètre est considérée comme impactée. Mais, dans le cas d'une méthode abstraite, en l'absence de corps à la méthode, cet impact ne peut être appliqué.

Le second élément à connaître est que les interfaces sont considérées comme des classes abstraites, étant donné qu'elles se comportent exactement de la même façon que celles-ci. L'implémentation d'une interface est donc considérée comme l'ajout d'un héritage.

Ce sont les seuls éléments à connaître pour bien comprendre l'interprétation d'une règle d'impact outre la nomenclature. Celle-ci a été mentionnée préalablement dans le présent chapitre, mais pour bien la comprendre, quelques exemples seront donnés ici, accompagnés de leur interprétation et d'explications.

Prenons la règle pour l'ajout d'une méthode : « Ma → Ma {Rr} (O : Sab) (H) + [Nam (L, A, H)] ». « Ma » est l'élément de changement. Ce qui suit la flèche « → » est la règle d'impact. Dans la règle d'impact, chaque élément impact est séparé par le signe d'addition « + ». Donc, pour l'ajout d'une méthode, on a deux éléments d'impact : l'ajout d'une méthode « Ma » et l'ajout d'appel à cette méthode « Nam ». Les crochets « [ ] » autour de l'élément d'impact « Nam » signifie l'incertitude. Ainsi, il pourrait y avoir des ajouts d'appels à la méthode. Pour l'élément d'impact « Ma », on retrouve à côté l'indication « Rr » entre accolade « { } ». Les accolades signifient une spécification de l'élément d'impact. « Rr » signifie « redéfinition ». Par conséquent, l'élément d'impact « ajout d'une méthode » sera une redéfinition de la méthode ajoutée. « O : Sab » indique que cet élément d'impact sera obligatoire « O » (certain) si la méthode ajoutée est abstraite « Sab », sinon facultatif (incertain). « (H) » et « (L, A, H) » correspondant respectivement aux éléments d'impact « Ma » et « Nam » sont les indicateurs de relation entre classes. Donc, on va chercher comme élément d'impact l'ajout d'une méthode dans les classes qui héritent « H » et comme élément d'impact l'ajout d'appel de la méthode dans les classes qui héritent « H », qui sont associées « A » et la classe elle-même « L » (local). Si on reprend une lecture complète de la règle, cela signifie que lorsqu'on fait l'ajout d'une méthode, il y aura comme impact l'ajout de redéfinition de cette méthode dans les classes qui héritent de façon certaine seulement si la méthode ajoutée est abstraite et qu'il y aura ajout d'appel à la méthode ajoutée de façon incertaine dans la classe elle-même, les classes associées et les classes qui héritent.

Comme second exemple, nous allons interpréter partiellement la règle d'impact d'un attribut qui passe de statique à non-statique : « At_{sn} → Mt_{sn} {Ru} (L) || Nra {Rms} (L) + Nra (A) ». L'élément d'impact « Mt_{sn} {Ru} (L) || Nra {Rms} (L) » signifie que les méthodes utilisant l'attribut dans la classe passeront de statique à non-statique

« Mt_{sn} » ou « || » on retira les utilisations de l'attribut dans les méthodes statiques « Nra » de la classe. On retrouvera aussi le signe « && » au lieu du « || » dans certaines règles, ce qui signifie que l'on retrouvera les deux changements dans l'élément d'impact, et non pas l'un ou l'autre comme dans cet exemple.

3.2.5.3 Lancement d'exception en Java

Les règles d'impact définies pour les lancements d'exceptions en Java ne s'appliquent pas à toutes les exceptions. Il existe dans les classes prédéfinies de Java une classe qui se nomme « RuntimeException ». Cette classe hérite de la classe « Exception » qui elle-même hérite de la classe « Throwable ». Cela signifie qu'un objet de la classe « RuntimeException » est une exception lançable. Tout ce qui est lançable (« Throwable ») respecte le comportement suggéré par les règles d'impact définies dans le modèle MICJ. Seulement, la classe « RuntimeException » modifie ce comportement en retirant l'obligation explicitement annoncée au niveau de la signature de la méthode que l'on va générer un certain type d'exception. Par ce fait, les règles du MICJ sur les lancements d'exception ne s'appliquent plus sur toutes les exceptions héritant de la classe « RuntimeException ».

Par exemple, si, dans une méthode *M1*, on ajoute à sa signature le lancement de l'exception « IOException » (classe d'exception prédéfinie de Java pour les erreurs liées aux entrées et sorties), cette dernière, héritant directement de la classe « Exception », doit être capturée par une structure « try/catch » ou relancée par toutes méthodes appelant *M1*. Par contre, si on ajoute à la signature de *M1* le lancement de l'exception « NumberFormatExeption » (classe d'exception prédéfinie de Java entre autre lancée lors du changement du type de donnée : « String » que l'on transforme en « Integer »), cette dernière, héritant de « RuntimeException », n'a plus à être capturée par une structure « try/catch » ou relancée par toutes méthodes appelant *M1*. En somme, les règles du modèle MICJ sur les exceptions ne s'appliquent pas aux classes d'exceptions héritant de « RuntimeException ».

3.2.6 Création d'une règle d'impact

L'élaboration du modèle MICJ et de ses règles d'impact découle d'un long processus de recherche et d'analyse.

La première étape a d'abord consisté à identifier tous les changements atomiques possibles du langage Java. Inspirés par les travaux de Chaumun, ces changements ont été divisés en trois niveaux : classe, méthode et attribut. Bien entendu, le niveau d'un changement dépend de la structure de la classe affectée par ce dernier : un attribut de classe, la méthode de la classe ou la classe. C'est ce qu'on appelle aussi des changements structurels (CS), puisqu'ils affectent la structure du programme. Comme le modèle de Chaumun est basé sur le langage C++, qui est aussi un langage orienté-objet, et qu'il a été adapté pour le langage Java par Kaibaili, la liste des changements atomiques est très similaire, sauf pour certains cas (héritage publique et privé possible en C++ mais pas en Java). Par contre, les changements liés au lancement d'exception ont été ajoutés. Les listes des changements atomiques structurels identifiés pour le modèle MICJ sont dans les tableaux III, IV et V pour respectivement la classe, la méthode et les attributs de classe.

Tableau III

Liste des changements atomiques structurels pour une classe.

Diminutif	Changement
Ca	Ajout d'une classe
Cr	Retrait d'une classe
Cn	Changement de nom d'une classe
Cia	Interface : ajout
Cir	Interface : retrait

Diminutif	Changement
Cha	Héritage : ajout
Chr	Héritage : retrait
Ct_{na}	Type : non-abstrait à abstrait
Ct_{an}	Type : abstrait à non-abstrait

Tableau IV

Liste des changements atomiques structurels pour une méthode.

Diminutif	Changement
Mv_{ui}	Visibilité : public à privée
Mv_{uo}	Visibilité : public à protégée
Mv_{iu}	Visibilité : privée à public
Mv_{io}	Visibilité : privée à protégée
Mv_{ou}	Visibilité : protégée à public
Mv_{oi}	Visibilité : protégée à privée
Mt_{na}	Type : non-abstrait à abstrait
Mt_{an}	Type : abstrait à non-abstrait
Mt_{sn}	Type : statique à non-statique
Mt_{ns}	Type : non-statique à statique
Mt_{fn}	Type : final à non-final
Mt_{nf}	Type : non-final à final

Diminutif	Changement
Mpa	Paramètre : ajout
Mpr	Paramètre : retrait
Mpc_t	Paramètre : changement de type
Mpc_n	Paramètre : changement de nom
Mwa	Throws : ajout
Mwr	Throws : retrait
Mr_{vo}	Retour : void à Object
Mr_{oo}	Retour : Object à Object
Mr_{ov}	Retour : Object à void
Ma	Méthode : ajout
Mr	Méthode : retrait
Mn	Méthode : changement de nom

Tableau V

Liste des changements atomiques structurels pour un attribut.

Diminutif	Changement
Av_{ui}	Visibilité: public à privée
Av_{uo}	Visibilité: public à protégée
Av_{iu}	Visibilité: privée à public
Av_{io}	Visibilité: privée à protégée
Av_{ou}	Visibilité: protégée à public
Av_{oi}	Visibilité: protégée à privée
At_{sn}	Type: statique à non-statique

Diminutif	Changement
At_{ns}	Type : non-statique à statique
At_{fn}	Type : final à non- final
At_{nf}	Type : non-final à final
Aa	Attribut : ajout
Ar	Attribut : retrait
An	Attribut : changement de nom
Ata	Changement type attribut

Pour écrire des règles d'impact précises et complètes, d'autres changements atomiques ont été identifiés. Certains impacts se retrouvaient à l'intérieur du corps des méthodes, plus précisément, au niveau des instructions. D'autres changements, comme le retrait d'un paramètre dans une méthode, vont forcer des changements à l'intérieur des méthodes. Dans cet exemple, on doit retirer les utilisations du paramètre au sein de la méthode. C'est ainsi, lors de la construction des règles d'impact, que la liste de changements atomiques dit non-structurels (CNS) (Tableau VI) a été identifiée:

Tableau VI

Liste des changements atomiques non-structurels.

Diminutif	**Changement**
Nad	Ajout de déclaration
Nrd	Retrait de déclaration
Nai	Ajout d'initialisation
Nri	Retrait d'initialisation
Nas	Ajout du mot clé « super »
Naah	Ajout appel attribut hérité
Namh	Ajout appel méthode hérité
Nrah	Retrait appel attribut hérité
Nrmh	Retrait appel méthode hérité
Nrs	Retrait du mot clé « super ».
Nrm	Retrait appel méthode
Nrcm	Retrait corps de la méthode (remplacé par « ; »)

Diminutif	**Changement**
Nacm	Ajout corps de la méthode (remplacement de « ; »)
Nrms	Retrait appel méthode statique
Nams	Ajout appel méthode statique
Natc	Ajout d'un « try/catch »
Nrtc	Retrait d'un « try/catch »
Nar	Ajout de « return »
Nrr	Retrait de « return »
Naa	Ajout appel attribut
Nra	Retrait appel attribut
Naf	Ajout instruction affectation
Nam	Ajout appel méthode
Nrf	Retrait d'instruction d'affectation

Au départ, il devait y avoir une règle d'impact pour chacun des changements atomiques ayant un impact qu'il soit ou non-structurel. En développant les règles, on s'est rendu compte que les règles d'impacts des CNS qui auraient pu en avoir étaient des redondances avec simplement une logique inversée. Par exemple, considérons le changement structurel « $\mathbf{M_{v0}}$ » d'une méthode qui ne retourne rien (« void ») à une

méthode qui retourne quelque chose (un objet quelconque). Un des impacts identifiable est « **Nr** », c'est-à-dire l'ajout de l'instruction « return » dans la méthode, qui est un changement non-structurel. Si on considère « **Nr** » comme changement initial, ce dernier aurait comme impact le passage de la méthode qui ne retourne rien à quelque chose. On se retrouve donc avec ces deux règles : « M_{vo} → Nr » et « Nr → M_{vo} ». Cela pourrait aussi s'écrire « M_{vo} ↔ Nr », signifiant une dépendance complète entre les deux changements. En conservant les deux règles telles quelles, cela signifie aussi que les deux changements sont des changements initiaux et des impacts en même temps, ce qui est illogique.

Dans tous les cas où un CNS peut avoir une règle d'impact, il y a le même genre de redondance. Pour corriger cette ambiguïté, on a décidé de considérer l'impact structurel comme changement initial. Ce faisant, aucun CNS n'a de règle d'impact.

Une fois les changements atomiques identifiés, il a été possible d'écrire les règles d'impacts. Ainsi, pour chacun des changements structurels, l'impact a été textuellement identifié.

Comme exemple, nous utiliserons le changement « retrait d'un attribut de classe » **Ar**. Quel impact aura-t-il ? Le retrait de l'attribut signifie qu'on doit retirer toutes ses utilisations, que son accesseur et son mutateur ne sont plus valides et doivent être retirés et que dans les constructeurs, s'il y avait des paramètres pour initialiser cet attribut, ils devraient être retirés, mais rien ne l'oblige.

Une fois qu'on possède l'information textuelle, on peut la transformer en une règle d'impact. Le changement initial est le retrait d'attribut : « Ar → ». Les impacts sont le retrait des utilisations : « Nra », le retrait de l'accesseur et du mutateur : « Mr {Rac} + Mr {Rmu} » et le retrait des paramètres dans le constructeur : « Mpr {Rc} ». On obtient donc la règle « Ar → Nra + Mr {Rac} + Mr {Rmu} + Mpr {Rc} ».

À cela on peut ajouter l'information sur le lieu : l'accesseur, le mutateur et les constructeurs sont locaux; le retrait des utilisations est local, dans les classes qui héritent, et dans les classes associées. On peut aussi ajouter la notion de certitude : le

retrait de l'accesseur, du mutateur et des utilisations de l'attribut sont certains ; le retrait des paramètres dans le constructeur est logique, mais facultatif et donc incertain. Il en résulte par conséquent la règle « Ar → Nra (L, A, H) + Mr {Rac} (L) + Mr {Rmu} (L) + Mpr {Rc} (L) ».

La dernière étape à la création d'une règle d'impact est sa validation à travers un exemple (code). On détermine donc une version du code initiale sur laquelle on appliquera le changement pour obtenir une version modifiée du code. Cette dernière doit contenir l'ensemble des impacts possibles.

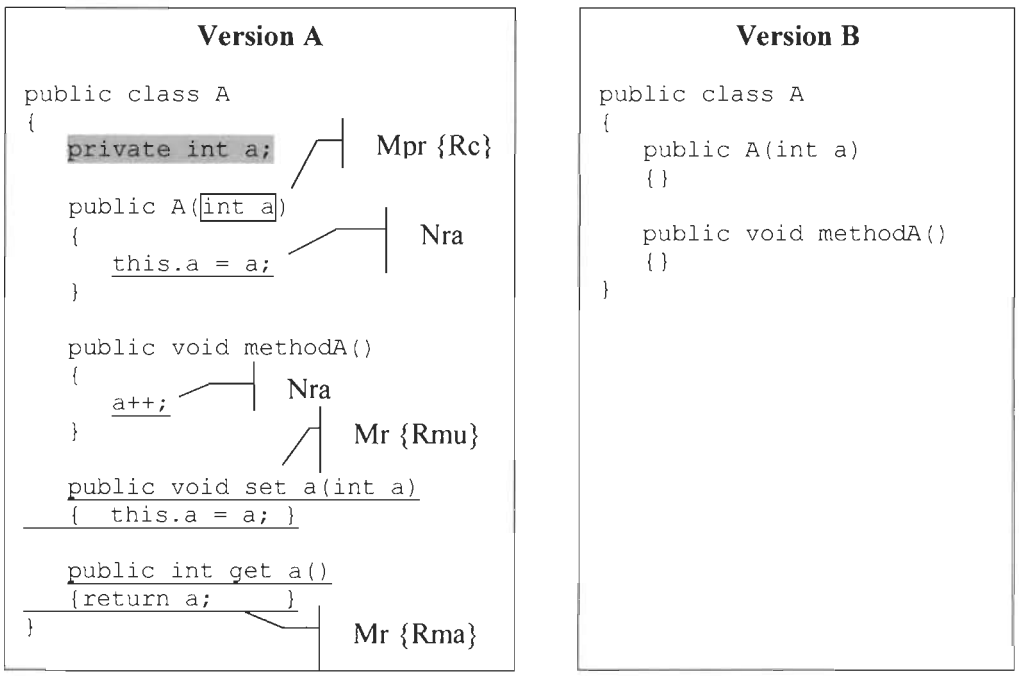

Figure 7 Code de validation d'une règle d'impact.

Dans la figure 7, nous avons dans une première version « A » un exemple de code qui est considéré comme code initial et une version « B » comme étant une version possible suite à l'application du changement qui, dans cet exemple, est le retrait d'un attribut. On a donc surligné en gris le changement initial (retrait de l'attribut « a »), souligné les impacts certains et encadré les impacts incertains.

Comme règle d'impact pour le retrait d'un attribut, on a la règle « Ar → Nra (L, A, H) + Mr {Rac} (L) + Mr {Rmu} (L) + Mpr {Rc} (L) ». Chaque impact est annoncé dans la figure 7.

L'ensemble des versions textuelles des règles d'impacts et leurs validations par le code est disponible à l'annexe C.

On constate que trois changements atomiques structurels ne sont pas présentés dans les annexes et dont les règles d'impacts ne sont pas disponibles. Il s'agit des changements de nom pour une classe, un attribut ou une méthode. La raison de cette absence n'est pas la difficulté d'écrire une règle d'impact et d'identifier ces impacts, mais plutôt la difficulté d'identifier ces changements de nom de façon automatisée. Pour remplacer ceci, on considère plutôt le retrait combiné à l'ajout. Un changement de nom est difficile à identifier, car entre deux versions, il peut y avoir un grand nombre de changements. En analysant le code, il est difficile de faire la distinction entre le fait d'avoir retiré une méthode dont la fonctionnalité est désuète pour la remplacer par une autre méthode portant un nom similaire et changer le nom d'une méthode et modifier son algorithme interne pour la rendre plus efficace par exemple. De plus, avec des environnements de travail performants tel que Eclipse, il est possible de faire du « refactoring » et donc de faire ce type de changement automatiquement. L'impact d'un changement de nom, se limitant au changement de l'identificateur, ne constitue pas une difficulté dans la prédiction de l'impact puisque ce changement n'affecte aucunement la logique du code sauf s'il y a conflit de nom.

CHAPITRE 4

ÉVALUATION EMPIRIQUE

4.1 Introduction

Plusieurs expérimentations ont été effectuées pour révéler les différences entre le modèle MICJ et celui de Chaumun. Elles ont également permis une évaluation globale du modèle MICJ en termes de capacité à faire une analyse prédictive. Pour cela, les expérimentations ont été divisées en plusieurs groupes. Le premier groupe d'expérimentations présente des résultats sur des changements unitaires ciblés pour rendre explicites les différences entre l'analyse avec le modèle de Chaumun et celle basée sur le modèle que nous proposons. Dans un second groupe d'expérimentations, les résultats ont été obtenus sur un seul changement effectué à répétition pour les deux modèles dans des programmes de petite taille. Dans le troisième groupe d'expérimentations, l'accent a été mis sur des résultats plus significatifs sur les capacités générales du modèle MICJ en effectuant une grande quantité de changements diversifiés dans plusieurs programmes. Ces premières évaluations expérimentales donnent dans l'ensemble des résultats très prometteurs. Il serait toutefois pertinent de poursuivre ces expérimentations dans le cadre de travaux futurs pour pouvoir tirer des conclusions solides et finales.

Les objectifs de ces premières expérimentations sont essentiellement de valider le modèle MICJ dans sa capacité à prédire l'impact correctement et de cerner aussi ses points forts ainsi que ses éventuelles limites. Pour quantifier la qualité des modèles évalués, nous utiliserons deux métriques introduites par Hassan et al. [25]: le rappel (« rappel ») et la précision (« précision »).

Lors des analyses, nous avons obtenu trois types de résultats: le nombre d'impacts réels suite à un changement, le nombre d'impacts prédits par un modèle et le nombre d'impacts prédits correctement par un modèle.

Le rappel est la capacité du modèle à prédire tous les impacts réels. Donc, le pourcentage d'impacts réels qu'il a prédits. Cela permet d'évaluer si les règles d'impact du modèle parviennent à prédire tous les impacts réels.

La précision est la capacité du modèle à prédire les impacts correctement. Donc, le pourcentage d'impacts prédits correspondant à la réalité. Cela permet de savoir si le modèle est suffisamment précis pour prédire des impacts réels. Cela permet aussi de valider le rappel, car si le rappel est très élevé par rapport à la précision, cela signifie que le modèle prédit trop d'impacts. Par exemple, avec le modèle de Chaumun, si on met comme règle d'impact pour tous les changements « H+A+G+L », donc toutes les classes qui ont un lien avec la classe où il y a le changement, nous obtiendrions à coup sûr un rappel très élevé, mais une précision très basse. Bref, le modèle nous donnerait des résultats non-significatifs.

Un modèle parfait serait donc un modèle permettant d'obtenir un rappel de 100% et une précision de 100%. Cela signifierait qu'il parvient à prédire uniquement des impacts réels et à tous les prédire.

4.2 Exemples ciblés

Les exemples ciblés sont une série d'expérimentations effectuées sur des changements atomiques sélectionnés pour leur capacité à avoir une analyse prédictive différente pour le modèle MICJ et le modèle de Chaumun. Chaque changement sera décrit avec détail pour permettre de comprendre comment l'analyse prédictive s'est effectuée afin de bien cerner les différences entre les deux modèles. Par la suite, pour chaque changement, nous procèderons à une analyse des résultats obtenus. Enfin, nous présenterons les résultats globaux de ces expérimentations.

4.2.1 Expérimentations

4.2.1.1 Exemple : Retrait d'attribut

Cet exemple met en valeur la précision du modèle MICJ par rapport à celui de Chaumun dans l'aide à la modification et à la prédiction de la charge de travail liée à une modification.

Dans cet exemple (voir figure 8), on veut passer d'une coordonnée tridimensionnelle à une coordonnée bidimensionnelle. On va donc retirer l'attribut « z » de la classe « Coordonnee ». Voici la règle d'impact du modèle de Chaumun correspondant au retrait d'attribut : S + L. Donc, il y aura un impact local et sur les classes associées.

Voici la règle d'impact du modèle MICJ : **Ar→**Mr {Rac} (L) + Mr {Rmu} (L) + Nra (L, H, A) + [Mpr {Rc} (L)]. Donc, le retrait d'un attribut implique le retrait de son accesseur, le retrait de son mutateur et le retrait des appels à l'attribut dans sa classe, dans les classes qui en héritent et dans les classes associées (si l'attribut est publique, cela peut se produire) ainsi que le retrait facultatif du paramètre du constructeur correspondant à l'attribut.

La figure 8 illustre le code des classes sur lesquelles on veut prédire l'impact lié au retrait de l'attribut surligné en gris :

```
public class Coordonnee
{
   protected int x;
   protected int y;
   protected int z;

   public  Coordonnee(int  x,
int y, int z)
   {
     this.x = x;
     this.y = y;
     thix.z = z;
   }

   public int getX()
   {  return x;  }
   public int getY()
   {  return y;  }
   public int getZ()
   {  return z ;  }

   public void setX(int x)
   {  this.x = x;  }
   public void setY(int y)
   {  this.y = y;  }
   public void setZ(int z)
   {  this.z = z;  }

   public void method()
   {
      y += z ;
   }
}
```

```
public class SuperCoor extends
Coordonnee
{
   private int t;

   public SuperCoor(int x, int
y, int z)
   {
      super(x,y,z);
      this.t = t;
   }

   public void methodZ()
   {
      z++;
   }
}
```

Figure 8 Version initiale.

Si on applique la règle du modèle de Chaumun, seule la classe « Coordonnee » sera impactée (sans aucune précision supplémentaire). La classe « SuperCoor » ne l'est pas, elle n'est ni associée ni locale.

Si on applique la règle d'impact du modèle MICJ, les impacts incertains sont encadrés et les impacts sont soulignés (voir figure 9). Le changement initial sera surligné en gris.

```
public class Coordonnee
{
   protected int x;
   protected int y;
   protected int z;

   public Coordonnee(int x,
int y, int z)
   {
     this.x = x;
     this.y = y;
     thix.z = z;
   }

   public int getX()
   {  return x;  }
   public int getY()
   {  return y;  }
   public int getZ()
   {  return z;   }

   public void setX(int x)
   {  this.x = x;  }
   public void setY(int y)
   {  this.y = y;  }
   public void setZ(int z)
   {  this.z = z; }

   public void method()
   {  y += z;     }
}
```

```
public class SuperCoor extends
Coordonnee
{
   private int t;

   public SuperCoor(int x, int
y, int z)
   {
      super(x,y,z);
      this.t = t;
   }

   public void methodZ()
   {
      z++;
   }
}
```

Figure 9 Version modifiée avec impact.

Pour notre modèle, si on observe l'impact au même niveau que le modèle de Chaumun (niveau classe), on a les classes « Coordonnee » et « SuperCoor » qui sont impactées.

La différence principale entre les deux modèles se voit dans leur précision. Le modèle de Chaumun ne fournit aucune autre information concernant la classe impactée. Si on souhaite évaluer l'effet en cascade avec le modèle de Chaumun, on doit absolument utiliser une autre technique suite à l'utilisation du modèle. Dans [3], les auteurs y vont de manière manuelle. Toute manipulation humaine est sujette à l'erreur, en particulier dans le cas de programmes complexes et de grande taille, surtout si celle-ci ne se base pas sur des règles précises. Le modèle MICJ a cet avantage en spécifiant de façon précise l'impact, et en appliquant les règles d'impact sur les impacts, on obtient l'analyse d'impact en cascade.

Dans cet exemple, l'impact se retrouve, donc, dans deux classes. Le modèle de Chaumun en a identifié une correctement sur un total d'une classe identifiée et le modèle MICJ en a identifié deux correctement sur un total de deux classes identifiées.

Le modèle MICJ a identifié trois (deux certains et un incertain) changements structurels (CS) comme impacts pouvant engendrer d'autres impacts et trois changements non-structurels (CNS) qui n'engendreront aucun impact.

4.2.1.2 Exemple : Attribut public en attribut privé

Nous allons ici utiliser l'exemple d'un attribut qui passe de public à privé. La règle d'impact du modèle de Chaumun est « S » ce qui signifie que les classes impactées sont les classes associées uniquement. Le modèle MICJ a pour règle d'impact « $\mathbf{Av_{ui}}$→Nra (H, A) + [Ma {Rmu} (L)] + [Ma {Rac} (L)] ». Il y aura, donc, impact dans les classes qui héritent et les classes associées par le retrait de l'attribut et s'il y a ajout de son mutateur et/ou de son accesseur, ce sera un impact dans la classe elle-même.

Voir l'exemple du code à la figure 10.

```
public class SuperCoor extends
Coordonnee
{
   private int t;

   public SuperCoor(int x, int t)
   {
      super(x);
      this.t = t;
   }

   public void methodZ()
   {
      x++ ;
   }
}
```

```
public class Coordonnee
{  public int x;  }

public class A
{
     private Coordonnee c;

     public A( )
     {
          c = new Coordonnee();
          c.x = 0;
     }
}
```

Figure 10 Exemple pour attribut publique à privé.

À la figure 10, l'attribut pour lequel on veut prédire l'impact s'il passe de public à privé est surligné en gris. Le modèle de Chaumun prédit l'impact uniquement dans les classes associées, soit la classe « A ». Le modèle MICJ va plutôt indiquer les classes « A » et « SuperCoor » comme étant des classes qui seront automatiquement impactées parce que la règle spécifie que le passage de l'attribut de publique à privé va obligatoirement impacter l'utilisation de l'attribut dans les classes qui héritent et les classes associées. De plus, la règle d'impact du modèle MICJ spécifie que comme l'attribut devient privé, il pourrait y avoir l'ajout de son accesseur dans la classe « Coordonnee », donc localement.

Le modèle de Chaumun ne permet donc pas d'aller chercher toutes les classes impactées avec certitude dans le cas d'un attribut de classe qui passe de public à privé. De plus, il ne donne aucune information concernant la nature de l'impact.

Dans cet exemple, l'impact se retrouve donc, dans deux classes avec certitude et une avec incertitude. Le modèle de Chaumun a identifié une classe avec certitude correctement sur un total d'une classe identifiée et le modèle MICJ en a identifié deux correctement sur un total de deux classes identifiées.

Le modèle MICJ a identifié un changement (incertain) structurel (CS) comme impact pouvant engendrer d'autres impacts et deux changements (certains) non-structurels (CNS) qui n'ont aucun impact.

4.2.1.3 Exemple : Attribut statique à non-statique

Pour le passage d'un attribut statique à non-statique, le modèle de Chaumun a comme règle d'impact « S+L », soit les classes associées et la classe elle-même. Le modèle MICJ a la règle d'impact « At_{sn} → Mtsn {Ru} (L) || Nra {Rms} (L) +Nra(A) », soit localement, les méthodes qui utilisent l'attribut vont passer de statique à non-statique ou bien on retire l'utilisation de l'attribut dans les méthodes statiques et on retire les utilisations de l'attribut dans les classes associées (possible quand l'attribut est public).

```
public class A
{
   private static int a;

   private static void aMethod()
   {
       a++;
   }
}

public class B
{
   private A b;
}
```

Figure 11 Exemple pour attribut statique à non-statique.

Dans cet exemple (voir figure 11), le changement est surligné en gris. Le modèle de Chaumun selon sa règle d'impact prédira que les classes « A » et « B » seront impactées alors que « B » ne le sera pas. Le modèle MICJ quant à lui prédira que la classe « A » sera impactée par l'un des deux changements en italique gras.

La différence entre les deux modèles, outre le détail de l'impact tel que soulevé lors des deux premiers exemples, est la surévaluation des classes par le modèle de Chaumun. En indiquant les classes impactées plutôt que le type d'impact, dans le cas de cette règle et de l'attribut qui est privé, il donne comme impact autant de classes associées qu'il y a, sans se soucier si celles-ci utilisent ou pas l'attribut modifié. Par ailleurs, comme aucune classe n'utilise l'attribut directement (attribut privé), aucune classe associée ne sera impactée.

Dans cet exemple, l'impact se retrouve, donc, dans une classe avec certitude. Le modèle de Chaumun a identifié une classe correctement sur un total de deux classes identifiées et le modèle MICJ en a identifié une correctement sur un total d'une classe identifiée (mais avec deux impacts).

Le modèle MICJ a identifié deux changements structurels (CS) comme impacts (l'un ou l'autre) pouvant engendrer d'autres impacts.

4.2.1.4 Exemple : Classe abstraite à non-abstraite

Le passage d'une classe abstraite à non-abstraite a pour règle d'impact selon le modèle de Chaumun « H+L », c'est-à-dire les classes qui héritent et la classe elle-même. Le modèle MICJ a la règle « $\mathbf{Ct_{an}}\rightarrow Mt_{an}$ {Rma} (L) + [Nai (L, A, H)] », c'est-à-dire que le passage d'une classe abstraite à non-abstraite a pour impact le passage des méthodes abstraites à non-abstraite de la classe elle même et l'ajout d'initialisation de l'objet dans les classes associées, celles qui héritent et dans la classe elle-même (localement).

La figure 12 illustre le changement que l'on veut prédire. Le changement est surligné en gris.

```
public abstract class A
{
   public abstract void methodA();
}

public class B
{
   public void methodB()
   {  A unA;      }
}
```

Figure 12 Exemple pour classe abstraite à non-abstraite

Selon le modèle de Chaumun, la classe A sera impactée sans toutefois spécifier que la méthode *A.methodA()* sera impactée comme le fait le modèle MICJ. De plus, le modèle de Chaumun qui est supposé inclure les impacts incertains ne prédit pas que la classe B pourrait être impactée comme le fait le modèle MICJ (impact encadré à la figure 12). Il faut comprendre ici que seules les classes associées (à la classe A dans

cet exemple) pourraient être impactées. Si, suite au changement, d'autres classes subissent des ajouts d'initialisations d'objet de la classe A, on ne pourra pas faire la distinction entre un simple changement d'algorithme qui demande l'utilisation de la classe A et le fait que ce soit le passage d'abstrait à non-abstrait qui permet cette utilisation.

Dans cet exemple, l'impact se retrouve, donc, dans une classe avec certitude et dans une autre classe avec incertitude. Le modèle de Chaumun a identifié une classe impactée avec certitude correctement sur un total d'une classe identifiée et n'a pas identifié la classe impactée avec incertitude. Le modèle MICJ a identifié une classe impactée avec certitude correctement et une classe impactée avec incertitude correctement sur un total de deux classes identifiées (une avec certitude et une avec incertitude).

Le modèle MICJ a identifié un changement structurel (CS) comme impact pouvant engendrer d'autres impacts et a identifié un changement non-structurel (CNS) pouvant se retrouver dans une classe.

4.2.1.5 Exemple : Méthode statique à non-statique

Nous allons maintenant faire l'analyse prédictive d'impact d'une méthode qui passe de statique à non-statique. Selon le modèle de Chaumun, l'impact est « I+L ». C'est-à-dire les invocations et localement. Selon le modèle MICJ, la règle d'impact est « $\mathbf{Mt_{sn}}\rightarrow$ Nrms (A, H) + [Nam (A, H)] », soit le retrait des appels statiques à la méthode et l'ajout d'appel (non-statique) à la méthode dans les classes associées et dans les classes qui héritent.

En se référant à la figure 13, le modèle de Chaumun spécifie par l'invocation, que la méthode *B.methodB()* sera impactée. C'est le seul lien « d'association » entre classes du modèle qui donne une information plus précise autre que la classe. Pour les besoins de l'analyse comparative des deux modèles, on considérera dans ce qui suit que le modèle de Chaumun spécifie la bonne classe. Le modèle de Chaumun indique aussi un impact local, donc la classe A. Le modèle MICJ indique que l'appel statique « A.methodA() » dans le corps de *B.methodB()* sera retiré, la classe B sera donc

impactée. De plus, il indique que la classe B pourrait être impactée par l'ajout d'appel non-statique de la méthode *A.MethodA()*. Pour les mêmes raisons que l'ajout d'initialisation de l'objet dans l'exemple précédent, même si ces ajouts d'appel peuvent être dans n'importe quelle classe, on considère seulement comme impact ceux ajoutés dans les classes associées. Le modèle MICJ ne le spécifie pas, mais dans le même ordre d'idée, on doit considérer comme impact l'ajout d'appel non-statique de la méthode seulement où il y a eu retrait d'appel statique de la méthode. Le modèle MICJ, contrairement au modèle de Chaumun, n'indiquera jamais la classe A (impact local) comme impactée.

```
public abstract class A
{
   public statique void methodA()
   {}
}

public class B
{
   public void methodB()
   {
      A.methodA();
   }
}
```

Figure 13 Exemple pour méthode statique à non-statique.

Dans cet exemple, l'impact se retrouve, donc, dans une classe avec certitude et avec incertitude. On ne fera pas ici la distinction puisque le modèle de Chaumun ne permet pas d'indiquer plus d'une fois une même classe. Le modèle de Chaumun a identifié une classe impactée avec certitude correctement sur un total d'une classe identifiée. Le modèle MICJ a identifié une classe impactée avec certitude et incertitude correctement sur un total d'une classe identifiée.

Le modèle MICJ a identifié un changement non-structurel (CNS) comme impact et a identifié un changement non-structurel (CNS) incertain pouvant se retrouver dans une classe.

4.2.1.6 Exemple : Changement de type d'un paramètre d'une méthode

Cet exemple traite du changement de type d'un paramètre d'une méthode. Il n'y a pas d'équivalence exacte entre les deux modèles. Au niveau du modèle de Chaumun, ce type de changement est considéré comme le changement de la signature d'une méthode soit pour une méthode abstraite ou non. Pour cette unique règle du modèle de Chaumun, il y en a trois dans le MICJ : l'ajout d'un paramètre, le retrait d'un paramètre et le changement de type d'un paramètre présenté ici en exemple pour une méthode abstraite. Le modèle de Chaumun a pour règle d'impact « H+L », c'est-à-dire que la classe elle-même et les classes qui héritent seront impactées. Le modèle MICJ a comme règle d'impact « **Mpc_t→** (Mpc_t || Ma) (O : Sab) {Rr} (H) + Nrm && Nam {Rh} (A, L) », c'est-à-dire les classes qui héritent, le changement de type du paramètre de la redéfinition de la méthode ou l'ajout d'une redéfinition de la méthode qui sera certain si la méthode est abstraite. De plus, il y aura le retrait et l'ajout de l'appel de la méthode dans les classes associées et localement si la hiérarchie des types n'est pas respectée. Par exemple, si pour une classe A existe la méthode *A.m(int i)* qui est appelée dans la classe associée B en passant un entier en paramètre (*int*) et que le changement de type du paramètre de *A.m()* est entier et passe à double précision (*double*), alors l'appel dans la classe B respecte la hiérarchie des types et ne sera pas impacté.

Dans cet exemple, on veut prédire l'impact pour le paramètre surligné en gris qui est double précision (*double*) qui deviendra entier (*int*) dans la figure 14.

Le modèle de Chaumun indique que les classes D et E seront impactées alors que le modèle MICJ indique que les classes C et E le seront. Dans cet exemple, contrairement à ce qu'indique le modèle de Chaumun, la classe D (local) ne sera pas impactée puisqu'on ne fait pas appel à la méthode. Le modèle MICJ indique dans la classe C que l'appel « t.methodD(3.0); » de *C.mothodC()* sera retiré car la hiérarchie des types n'est pas respectée dans ce cas. Il indique aussi que dans la classe E, on doit soit changer le type du paramètre de la méthode *E.methodD(double)*, soit ajouter une redéfinition de la méthode *D.methodD(int)*.

```
public class C
{
   public void methodC()
   {
      D[] t = new D[10];
      t.methodD(3.0);
   }
}

public abstract class D
{
   public abstract void methodD(double val);
}

public class E extends D
{
   public void methodD(double val)
   {}
}
```

Figure 14 Exemple pour changement de type d'un paramètre.

Dans cet exemple, le modèle de Chaumun ne fait pas fausse route sauf sur le fait qu'il n'inclut pas les classes associées. Il a raison de cibler l'héritage et localement même si dans l'exemple l'impact n'est qu'au niveau de l'héritage. Si la méthode était appelée localement, il y aurait effectivement impact. Mais encore une fois, la différence majeure entre les deux modèles, est la précision par rapport à l'impact. Dans le modèle MICJ, en spécifiant la nature de l'impact, on permet d'éviter de considérer toutes les classes liées à la classe en question. De plus, il est facile de modifier la règle d'impact dans le modèle MICJ pour éviter d'arriver à des résultats non conformes aux attentes.

Dans cet exemple, l'impact se retrouve donc, dans deux classes avec certitude. Le modèle de Chaumun a identifié une classe impactée correctement sur un total de deux classes identifiées. Le modèle MICJ a identifié deux classes impactées correctement sur un total de deux classes identifiées.

Le modèle MICJ a identifié deux changements structurels (CS) comme impacts pouvant engendrer d'autres impacts.

4.2.1.7 Exemple : Lancement d'exception d'une méthode

L'ajout ou le retrait d'un lancement d'exception à une méthode sont les deux changements dont l'impact est le plus important. Celui-ci provient du fait que le lancement d'exception affecte toutes les méthodes appelant directement ou indirectement celle qui lance cette exception. Toutes les méthodes qui appellent ou qui sont redéfinies par une méthode qui lance une exception se doivent de relancer ou de traiter l'exception. Donc, si le couplage est fort et que l'exception n'est pas traitée, l'ajout ou le retrait d'un lancement d'exception a un effet en cascade important.

A cet effet, le modèle MICJ défini des règles d'impact pour l'ajout et le retrait de lancement d'exception à une méthode. Le modèle de Chaumun n'a aucune règle associée au lancement d'exception ce qui peut être considéré comme une lacune relativement importante.

Puisque le modèle de Chaumun ne possède aucune règle pour comparer avec le modèle MICJ, cet exemple ne sera pas pris en considération pour le paragraphe 5.1.2 dans les calculs.

4.2.2 Résultats

L'analyse des résultats qui suit se base sur les exemples donnés précédemment. Il est important de comprendre que ces exemples ont été donnés dans l'objectif de souligner la différence de l'analyse entres les deux modèles. Les valeurs obtenues sont explicites sur l'apport du modèle MICJ en précision par rapport au modèle de Chaumun. Ces valeurs ne reflètent cependant pas la différence aux niveaux des valeurs de cas réels si on reste seulement à un niveau de précision des classes. Bref, on présente donc une analyse qualitative plutôt que quantitative.

Dans le tableau VII, la valeur « I » donne le nombre de classes impactées réellement. Pour chacun des modèles, on a la valeur « CT » qui est le nombre total de classes identifiées et la valeur « CC » qui est le nombre de classes correctement identifiées. La métrique « rappel » est un indice de qualité du modèle. Elle mesure la proportion de classes impactées que le modèle a correctement identifiées par rapport à la réalité.

La métrique « préc. » est un indice pour mesurer la précision du modèle dans les classes impactées qu'il a identifiées. Elle mesure le nombre de classes impactées identifiées correctement par le modèle par rapport au nombre total de classes réellement impactées.

Le tableau VII considère une analyse stricte, c'est-à-dire que l'on considère seulement les classes impactées réellement dont on peut être certain qu'elles seront impactées.

Tableau VII

Résultats de l'expérimentation.

I	Chaumun				MICJ			
	CC	CT	Rappel	Préc.	C	CT	Rappel	Préc.
9	6	8	67 %	75 %	9	9	100 %	100 %

Donc, le modèle de Chaumun correspond à 67% de la réalité et l'information qu'il donne est fiable à 75% par rapport à des valeurs parfaites du modèle MICJ.

Le tableau VIII présente une large analyse des résultats, conformément au modèle de Chaumun qui couvre les impacts incertains, on prend dans cette analyse en considération les classes impactées avec incertitude.

Tableau VIII

Résultats de l'expérimentation.

I	Chaumun				Modèle d'impact de changement en Java			
	CC	CT	Rappel	Préc.	C	CT	Rappel	Préc.
11	6	8	55 %	75 %	11	11	100 %	100 %

On constate que l'effet se manifeste uniquement sur le « rappel » du modèle de Chaumun. Cela signifie qu'en prenant en compte les classes impactées avec incertitude, le modèle de Chaumun correspond encore moins à la réalité (55%).

Il est important de mentionner que le modèle MICJ a pu identifier les impacts avec plus de précision que le modèle de Chaumun. Le modèle de Chaumun se limite

uniquement à identifier les classes impactées contrairement au modèle MICJ qui a réussi à identifier 9 changements structurels (CS) et 8 changements non-structurels comme impacts, fournissant ainsi plus d'informations et plus de détails. De plus, ce sont les CS qui permettent de faire l'analyse en cascade et donc de pousser plus loin l'analyse d'impact automatisée. Le modèle de Chaumun, ne donnant aucune information sur la nature de l'impact ne permet pas de faire ce type d'analyse.

Pour résumer, la différence majeure entre les deux modèles se situe particulièrement au niveau de la granularité du modèle. Le modèle MICJ, en spécifiant la nature de l'impact, permet de filtrer les classes liées à la classe à modifier. Ce filtre fait toute la différence en terme de précision du modèle (valeur « préc. » dans les tableaux). Le fait de spécifier la nature des impacts, de fractionner l'impact, permet d'assouplir le modèle et d'être plus proche de la réalité ce qui augmente le « rappel ». En cours de l'expérimentation, si un impact n'a pas été couvert, il sera possible d'ajuster la règle d'impact pour améliorer la qualité du modèle MICJ (« rappel ») sans nuire à sa précision (« préc. »). Ceci ne peut être fait par le modèle de Chaumun. Modifier une règle pourrait dans certain cas augmenter son « rappel », mais nuirait alors à sa « préc. ».

4.3 Autres Expérimentations

Ces expérimentations sont une suite d'analyses d'impacts sur un seul changement effectué à plusieurs reprises dans plusieurs programmes similaires. L'objectif de telles expérimentations est de vérifier la stabilité du modèle dans la mesure où des petites différences d'implémentation ne devraient pas affecter l'analyse prédictive du modèle sauf si celles-ci impliquent l'élément changé et ses impacts. Dans les expérimentations précédentes, on a pu démontrer, entre autres, que le modèle de Chaumun manque parfois de précision à cause de ses règles qui sont plus générales que celles du modèle MICJ. On peut supposer que si le modèle MICJ a des règles d'impacts précises et moins générales, ses résultats restent similaires d'un programme à un autre.

4.3.1 Expérimentation

Cette expérimentation a été effectuée sur des programmes développés par des étudiants, dans le cadre d'un cours. Nous avons choisi six programmes sur vingt. Le choix s'est basé sur la note (six meilleures notes). Ces programmes vont probablement respecter le même schéma de code (comparables), ce qui permet une analyse prédictive facile.

L'objectif du devoir est de réaliser une classe *Vivant* abstraite, trois classes qui héritent de celle-ci (*Animal*, *Humain* et *Plante*) et une classe pilote pour tester les fonctionnalités. La classe *Vivant* a un indice de complexité et une exception qui doit être lancée lorsqu'on tente de créer un objet *Vivant* avec une complexité inférieur à zéro ou supérieur à 10. De plus, la classe *Vivant* doit avoir comme attribut le nom et le milieu de vie.

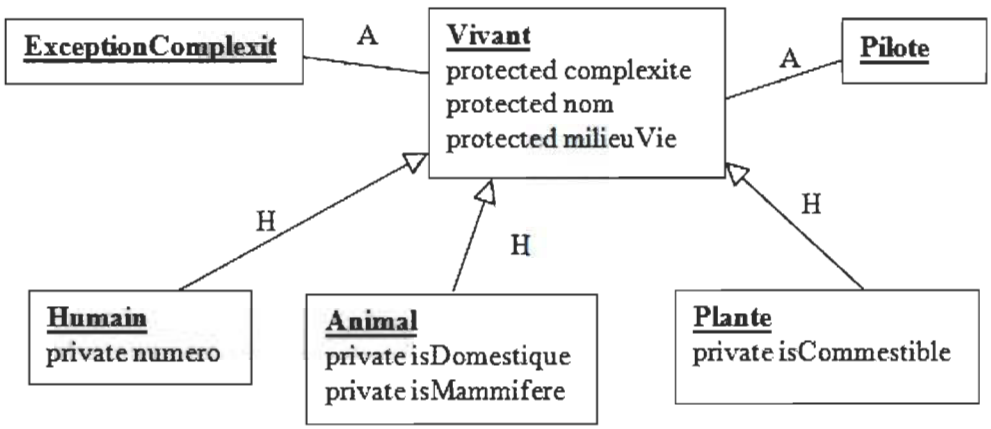

Figure 15 Diagramme de classes du devoir.

La figure 15, montre le diagramme de classes qui est respecté par l'ensemble des travaux sélectionnés :

Pour chacun des travaux, on a testé le retrait des attributs de la classe *Vivant*. Donc, nous avons effectué une prédiction de l'impact sur le retrait de trois ou quatre attributs pour chaque programme. Selon le modèle de Chaumun, l'impact est « S+L », c'est-à-dire les classes associées et la classe locale. Donc, dans tous les cas

c'est la classe *Vivant* et la classe pilote qui sont impactées. Le modèle MICJ a comme règle d'impact « Mr {Rac} (L) + Mr {Rmu} (L) + Nra (L, H, A) + [Mpr {Rc} (L)] ». Donc, le retrait de l'accesseur et du mutateur dans la classe *Vivant*, le retrait des utilisations de l'attribut (*Nra*) dans la classe *Vivant*, les classes qui en héritent, la classe pilote et le retrait du paramètre dans les constructeurs de la classe *Vivant*. Ce dernier impact est dit facultatif car normalement, en retirant l'attribut, le paramètre du constructeur lui correspondant devrait lui aussi être retiré s'il y en a un. Par contre, il peut ne pas être retiré et cela n'empêchera pas le programme d'être correct syntaxiquement et de continuer à fonctionner normalement.

4.3.2 Résultats

Les résultats globaux pour cette expérimentation sont présentés dans le tableau IX.

Tableau IX

Résultats de l'expérimentation.

Nombre de changements	Prévision		Réel	Détail MICJ		Nombre classe total
	MICJ	Chaumun		Certains	Incertains	
21	35	42 / 20	35	79	18	39

	Précision	Rappel
Chaumun	48 %	57 %
MICJ	100 %	100 %

Le tableaux IX indique que l'on a testé 21 retraits d'attributs. Le modèle MICJ a identifié 35 classes impactées qui correspondent aux 35 classes réellement impactées. Le modèle de Chaumun a identifié 42 classes impactées dont 20 correspondent au classes réellement impactées. Parmi les 35 classes impactées identifiées par le modèle MICJ, celui-ci a identifié 79 impacts certains et 18 qui pourraient avoir lieu (retrait de paramètre dans les constructeurs). Dans l'ensemble des programmes, il y avait 39 classes.

En raison du manque de détails dans le modèle de Chaumun, les valeurs des métriques sont très faibles. Il ne spécifie pas la nature de l'impact dans les classes ciblées et elles ne sont donc pas filtrées. Dans tous les programmes, il a identifié la

classe pilote comme étant une classe impactée ce qui pourrait effectivement être le cas lorsqu'il y a des attributs publiques. Mais les rares attributs publiques qu'il y a eu dans les devoirs n'ont pas été utilisés via la classe pilote. Dans sa façon d'exprimer l'impact en spécifiant l'utilisation de l'attribut dans les classes associées, le modèle MICJ permet de ne pas cibler la classe pilote puisque celle-ci n'utilise pas les attributs. De plus, le modèle de Chaumun ne considère aucun impact au niveau des classes qui héritent, ainsi dans la plupart des programmes qui ont servis à l'expérimentation, il omet d'indiquer les classent qui héritent et qui sont impactées. Une omission que ne fait pas le modèle MICJ.

Il est à noter que même si la règle d'impact de Chaumun aurait été « S+L+H » (donc, avec l'ajout des classes qui héritent), cela aurait eu un impact positif dans le cas de 5 des 6 programmes utilisés seulement. Dans le sixième, les attributs de la classe *Vivant* sont privés au lieu de protégés. Leurs utilisations dans les classes qui héritent passent donc par les accesseurs et mutateurs ce qui fait que le retrait des attributs n'a pas un impact direct dans les classes qui héritent. De plus, même si les attributs étaient protégés, cela ne garantit pas leur utilisation dans les classes qui héritent ce qui pourrait causer une baisse de précision dans le modèle de Chaumun.

Les résultats de 100% de précision et de rappel pour le modèle MICJ ne sont pas significatifs pour l'ensemble des règles du modèle. Par contre, ils nous assurent que malgré les petites différences d'implémentations des programmes, la règle d'impact est toujours parvenue à bien prédire l'impact et de façon constante. Il est donc possible de dire que le comportement de la règle d'impact est stable, car elle n'est pas influencée par les détails d'implémentation.

4.4 Expérimentations sur les patterns

Précédemment, les deux expérimentations ont été faites en ciblant des forces du modèle MICJ et des faiblesses du modèle de Chaumun pour faire ressortir la différence créant ainsi un écart volontaire entre les résultats obtenus par les deux modèles. L'écart étant volontaire et provoqué, il n'est donc pas tout à fait significatif. L'expérimentation suivante propose donc une série de changements diversifiés dans plusieurs programmes différents (avec des Patterns) .

L'objectif est bien sûr d'obtenir des résultats significatifs sur les capacités des deux modèles et d'obtenir de nouveaux éléments d'analyse sur les forces et faiblesses du modèle MICJ.

4.4.1 Expérimentation

Pour cette expérimentation, nous avons utilisé des programmes représentant des patterns. Les programmes varient en nombre de classes : entre 4 et 10 classes. Au total, 40 changements ont été testés. Pour obtenir des données réelles, les changements ont ensuite été implémentés. Cela permet de tester les modèles avec différents types de changements.

Dans certains cas, il y a eu des effets en cascade dans les modifications. Pour les besoins de la comparaison entre les deux modèles et parce que le modèle de Chaumun ne permet pas de gérer les impacts en cascade, chaque impact sur lequel on peut faire une analyse en cascade a été traité comme un nouveau changement.

Dans cette expérimentation, on obtient les résultats avec le niveau de précision de Chaumun, c'est-à-dire en spécifiant les impacts au niveau de la classe : quels sont les classes qui seront impactées. Le modèle MICJ, même s'il indique plus d'un impact dans une classe, il sera considéré comme un seul impact car une seule classe impactée.

Les patterns considérés sont les suivants : *abstract factory, adapter, bridge, builder, chain of responsibility, command, composite, decorator, façade, interpreter et momento.*

4.4.2 Résultats

Le modèle de Chaumun a trouvé un total de 46 classes impactées pour les 40 changements considérés : 30 de ces impacts correspondent à des impacts réels et 16 sont erronés. Le modèles MICJ à trouvé 71 classes impactées pour les 40 changements : 49 sont des impacts que le modèles MICJ prédit comme étant certains et correspondent tous aux impacts réels. Les 24 autres impacts sont des impacts que le modèle MICJ prédit comme étant incertains dont 14 correspondent à des impacts

réels. En réalité, il y a 71 classes qui ont été impactées pour les 40 changements. Les résultats de cette expérimentation sont disponibles dans le tableau X

Tableau X

Résultats de l'expérimentation.

Nombre de changement	Prévision		Réalité
	MICJ	Chaumun	
36	41/41 – 14/30	27/41	57

	Précision	Rappel
Chaumun	65,85%	47,37%
MICJ	100,00% - 77,46%	96,49%

Au niveau du « rappel », qui correspond à la capacité du modèle à identifier tous les impacts, le modèle MICJ a obtenu un taux de 96% alors que le modèle de Chaumun a obtenu un taux de 47%. Au niveau de la précision qui correspond à la capacité du modèle à prédire les impacts réels, le modèle de Chaumun a obtenu un taux de 66% alors que le modèle MICJ a obtenu un taux de 100% en considérant uniquement les prédictions certaines. En considérant aussi les prédictions incertaines, on obtient plutôt une précision de 77%.

Grâce à cette expérimentation, on constate que le modèle MICJ n'est pas parfait dans sa prédiction d'impact et que globalement, même en choisissant plusieurs types de changements à prédire, le modèle de Chaumun reste avec une « précision » et un « rappel » inférieur au modèle MICJ. De plus, l'imperfection au niveau de la précision du modèle MICJ se situe uniquement au niveau des impacts incertains. Ils présentent effectivement une précision de 47% (14 des 30 impacts incertains correspondent à la réalité).

Dans cette expérimentation, de nombreuses imperfections du modèle MICJ ont été identifiées l'empêchant ainsi de prédire correctement les impacts et d'avoir une précision de 100%. On discutera de cela dans la section 4.7.

4.5 Expérimentations du MICJ non-comparatives

Les expérimentations du modèle MICJ non-comparatives ont pour objectif d'obtenir des données précises sur les capacités du modèle. Jusqu'à maintenant, pour les besoins de la comparaison, les résultats ont été obtenus selon le niveau de précision du modèle de Chaumun qui est dit « de classe ». Donc, on regarde simplement si les modèles prédisent les bonnes classes comme étant impactées. Par contre, le modèle MICJ permet un niveau de précision plus élevé que celui de Chaumun. Il faut donc, pour mesurer correctement ces niveaux de précision et de rappel, considérer d'autres changements avec le modèle MICJ.

L'objectif étant d'obtenir des résultats qui évaluent le modèle MICJ en tant que tel et sans le comparer à un quelconque modèle.

4.5.1 Expérimentations

Dans l'expérimentation de la section 4.4, les résultats du modèle MICJ on été obtenus en fonction de la précision du modèle de Chaumun, c'est-à-dire que nous avons considéré tout impact du modèle MICJ comme un impact de niveau classe sans considérer la multiplicité. Donc, si on a un changement avec un impact sur trois instructions dans le corps de trois méthodes différentes d'une classe, on considère seulement un impact qui est celui de la classe puisque le modèle de Chaumun ne permet pas d'avoir une évaluation plus précise et ne fait aucune multiplicité d'impact au sein d'une classe (niveau de granularité limité aux classes).

Dans le tableau XI, on présente les résultats avec plus de détails sur le modèle MICJ par rapport à la réalité en considérant la même expérimentation effectuée à la section 4.4, c'est-à-dire les 11 patterns utilisés.

4.5.2 Résultats

Les résultats sont donc les suivants : les impacts que prédit avec certitude le modèle MICJ ont toujours été exacts par rapport à la réalité sauf dans un cas (voir section 4.7.1.3). C'est dans les impacts incertains que l'on retrouve des prédictions erronées. Seulement 32% des impacts prédits avec incertitude se sont réellement produit. Les impacts incertains représentent près de la moitié des impacts prédits au

total (56 sur 107). La totalité des mauvaises prédictions sont donc faites par les prédictions d'impacts incertains et une seule par les prédictions d'impacts certains. C'est 26% des impacts que le modèle MICJ a prédit correctement qui a été prédit par les impacts incertains (18 des 68 au total).

Tableau XI

Résultats de l'expérimentation.

Nombre de changements	MICJ	
	Prévision	Réel
35	50/51 – 18/56	71

Précision	Prec incertains	Rappel
98,04% - 63,55%	32,14%	95,77%

Le modèle de Chaumun spécifie qu'il prédit les impacts incertains. Dans cette idée, si on combine les résultats pour les impacts certains et incertains du modèle MICJ, on obtient une précision de 64%, soit 2% de moins que le modèle de Chaumun pour les mêmes tests dans les expérimentations précédentes. Il faut néanmoins comprendre que l'imprécision vient de l'incertitude et que le modèle MICJ permet de faire cette distinction. Donc, globalement, il n'est pas plus précis, mais il permet d'identifier ce qui est incertain contrairement au modèle de Chaumun qui combine le tout. Cet écart de précision n'est pas si grand et pourrait varier selon les tests effectués. Le modèle MICJ présente l'avantage de faire la distinction entre les impacts certains et incertains. En plus d'être précis, il apporte donc cette nuance importante sur les impacts.

Si le modèle MICJ n'est pas globalement plus précis que le modèle de Chaumun, il présente toutefois un « rappel » beaucoup plus élevé. Soit un « rappel » de 96% pour le modèle MICJ et de 47% pour le modèle de Chaumun. Cela signifie que le modèle MICJ permet d'identifier presque la totalité des impacts réels et le double d'impacts réels identifiés par le modèle de Chaumun.

On peut donc dire que malgré la précision globale du modèle MICJ similaire au modèle de Chaumun, le modèle MICJ reste supérieur dans sa capacité à identifier les

impacts réels en plus de fournir davantage d'informations comme la nature de l'impact qui permet de faire de l'analyse en cascade et permet, à la lumière des expérimentations, de savoir de façon certaine ou incertaine là ou il y aura un impact.

4.6 Expérimentations supplémentaires

Ces expérimentations supplémentaires ont été faites sur un autre groupe de programmes et visent aussi à évaluer objectivement le modèle MICJ. Tout comme les expérimentations précédentes, les résultats sont obtenues selon la granularité des règles du modèle MICJ. Cela permet aussi d'apporter de nouvelles situations de changement pour valider le modèle.

4.6.1 Expérimentations

Pour cette nouvelle série d'expérimentations, nous avons analysé huit programmes possédant un minimum de deux versions et un maximum de six. Au total, c'est 31 versions de divers programmes qui ont été considérées, ayant deux à vingt classes chacune. À travers ces différentes versions, c'est 31 changements qui ont été considérés dont l'impact a été d'abord prédit et ensuite comparé avec les impacts réels.

4.6.2 Résultats

Les résultats obtenus sont disponibles dans le tableau XII.

Tableau XII

Résultats de l'expérimentation.

Nombre de changement	Prévision MICJ		Réel
	Certain	Incertain	
31	43 / 68	28 / 58	104

Précision			Rappel
Globale	Certain	Incertain	
56,35%	63,24%	48,28%	68,27%

Pour les 31 changements, il y a eu un total de 104 impacts dans le code. Le modèle MICJ a prédit 68 impacts avec certitude dont 43 impacts réels. Il a aussi prédit 58 impacts avec incertitude dont 28 impacts réels. Au total, le modèle MICJ a donc prédit 126 impacts dont 71 impacts réels. Sa précision est de 56% et son rappel est de 68%.

Ces expérimentations donnent des résultats pour la précision et le rappel significativement plus bas que les expérimentations sur les patterns. Cela pourrait porter à croire qu'il y a eu un problème dans le protocole ou que l'une des deux expérimentations n'est pas représentative. Mais en analysant de plus près les données, on constate que c'est plutôt lié à un groupe de changements de ces expérimentations qui ont eu beaucoup d'impacts. D'ailleurs, en les retirant de l'analyse on obtient une augmentation globale de 6% de la précision, de 17% de précision pour les impacts prédits avec certitude, et de 12 % pour le rappel. Parmi ces trois impact, l'un d'eux est l'ajout d'une classe impact non prévu par notre modèle. Ce changement d'ajout de classe est responsable à lui seul de 12 impacts réels que le modèle est incapable de prédire sur les 104 impacts réels au total. Dans les changements, en tout et pour tout, les quatre changements ajout d'une classe sont responsables de 22 impacts réels que le modèle ne prédit pas. Ces impacts non prédits représentent environ 21% des impacts. Dans ces cas, la modèle MICJ ne pourra pas avoir un rappel de plus de 79%. On peut alors considérer qu'un rappel de 68% pour l'ensemble des changements effectués est significativement élevé.

La précision, relativement basse pour la prédiction des impacts certains, et par conséquent la précision globale de ces expérimentations, est quant à elle totalement attribuable à un groupe de cinq changements ayant eu lieu entre deux versions d'un même programme et pour lesquels le modèle MICJ prédit des impacts certains à tort.

Des expérimentations supplémentaires seront nécessaires pour vérifier ce genre de cas de figures.

4.6.3 Résultats combinés des expérimentations

Les résultats de la dernière série d'expérimentations et de celles sur les patterns ont été combinés puisque les deux respectent le même protocole. Cela permet aussi d'obtenir des résultats plus généraux sur le modèle MICJ. Des éléments de corrélation ont aussi été extraits de cette analyse, essentiellement entre ce que prédit le modèle et ce que nous pouvons observer dans la réalité (impacts réels après changement). Les résultats sont disponibles dans le tableau XIII.

Tableau XIII

Résultats de l'expérimentation.

Nombre de changement	Prévision MICJ		Réel
	Certain	Incertain	
66	93 / 119	46 / 114	175

Précision			Rappel
Globale	Certain	Incertain	
59,66%	78,15%	40,35%	79,43%

Pour un total de 66 changements, il y a eu 175 impacts réels. Le modèle MICJ a prédit 119 impacts avec certitude dont 93 impacts réels. Il a aussi prédit 114 impacts incertains dont 46 impacts réels. On a obtenu une précision globale de 60%, une précision pour les impacts prédits avec certitude de 78% et un rappel de 79%. Le modèle MICJ a donc prédit correctement 139 des 175 impacts. Cependant, la précision pour les impacts prédits avec incertitude est très faible et beaucoup plus basse que la précision des impacts prédits avec certitude, soit de 40% et presque deux fois moins élevée que les impacts prédits avec certitude.

Dans ces expérimentations, le modèle permet d'obtenir de bons résultats de prédiction d'impacts avec un rappel de 79% qui signifie qu'il a prédit 79% des impacts réels avec succès. C'est beaucoup plus que ce que le modèle de Chaumun est parvenu à faire dans les expérimentations sur les patterns avec un rappel de 47%. De plus, le modèle MICJ obtient ce résultat avec une granularité de l'impact beaucoup plus fine et plus précise que celle du modèle de Chaumun. Il prédit donc avec plus d'efficacité et plus de détails l'impact du changement.

Avec ces expérimentations, nous avons aussi voulu vérifier, si certaines corrélations existent entre les changements, la taille de leur impact et les prédictions du modèle MICJ. Le tableau XIV donne les facteurs de corrélation entre divers éléments :

Tableau XIV

Résultats de l'expérimentation.

	Correct+erreur	Correct	Erreur
IR/IP C	0,56	0,63	0,22
IR/IP I	0,19	0,42	-0,09
IR/IP Tot	0,60	0,81	0,07

Pour effectuer une analyse des corrélations (valeurs fournies dans le tableau XIV), nous utilisons le coefficient de corrélation linéaire empirique aussi appelé **r de Pearson** [34], [35], [36]. Ce coefficient se calcule pour *n* paires d'observation (x_i, y_i). Le calcul de **r** est effectué par :

$$r = \frac{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)\left(y_i - \bar{y}\right)}{\sqrt{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}\sqrt{\sum_{i=1}^{n}\left(y_i - \bar{y}\right)^2}} \qquad (1)$$

où **r** est indépendant des choix d'unités de mesure de **X** et **Y**. Sa valeur varie entre 1 et -1. Pour une corrélation linéaire empirique parfaite, on obtiendra $r = 1$. Cela signifie que **Y** augmente parfaitement linéairement par rapport à **X**. Donc, que **Y** est parfaitement proportionnel à **X**. Inversement, si on obtient $r = -1$, cela signifie que **Y** diminue parfaitement linéairement par rapport à **X**. Donc, que **Y** est parfaitement inversement proportionnel à **X**. Si on obtient $r = 0$, cela veut uniquement dire qu'il n'y a aucune corrélation linéaire entre **X** et **Y**. Cela ne veut pas dire qu'il n'y a aucune forme de corrélation.

La première colonne, « Correct + erreur », considère le nombre d'impacts prédits correctement et incorrectement, la seconde colonne donne le nombre d'impacts prédits correctement et la troisième colonne donne le nombre d'impacts prédits

incorrectement. La première rangée donne la corrélation entre le nombre d'impacts réels (**IR**) et le nombre d'impacts prédits (**IP**) avec certitude (**C**) par le modèle en fonction de ce qui est considéré à chaque colonne pour chaque changement. La seconde rangée donne la corrélation entre le nombre d'impacts réels et le nombre d'impacts prédits avec incertitude (**I**) pour chaque changement. La dernière rangée donne la corrélation entre le nombre d'impacts réels et le nombre d'impacts prédits au total (**Tot**) par le modèle MICJ pour chaque changement.

A la lecture de ces résultats, on peut voir qu'il n'y a pas de lien direct entre le nombre d'impact réels et le nombre d'impact prédits avec incertitude, et ce, tout particulièrement pour les impacts prédits incorrectement. Cela signifie que le nombre d'impacts prédits avec incertitude est un mauvais indice pour connaître la taille réelle de l'impact. On peut aussi constater qu'il y a une forte corrélation entre le nombre d'impacts réels et le nombre d'impacts prédits correctement par le modèle MICJ. Cela indique que les règles d'impacts du modèle sont toutes en général capables de prédire correctement les impacts réels.

4.7 Discussion sur les modèles et les résultats

Comme dans chacune des expérimentations effectuées avec les deux modèles, le modèle MICJ a montré que par son niveau de détail, il parvient à obtenir des prédictions avec une meilleure précision et un meilleur rappel que le modèle de Chaumun. De plus, le modèle MICJ fournit des informations complémentaires. C'est-à-dire qu'en plus de dire quelles classes seront impactées, il apporte d'autres précisions (comment et où). Ces informations peuvent permettre de faire un calcul d'impact en cascade automatisé et d'évaluer la charge de travail liée à un changement.

4.7.1 Limites du modèle MICJ

Les diverses expérimentations effectuées montrent que le modèle MICJ obtient des résultats proches de la réalité et fournit une très grande quantité d'informations. Néanmoins, celles-ci ont aussi montré que le modèle présente quelques limites pour le calcul de certains impacts comme par exemples lorsqu'il y a de l'héritage ou lors

de l'ajout d'une classe. Le but de ces expérimentations, au-delà de l'évaluation de ses capacités, étant aussi d'identifier ses limites éventuelles pour pouvoir mieux orienter certains travaux futurs.

4.7.1.1 Héritage

Dans le modèle MICJ, on considère l'appartenance des méthodes et des attributs aux classes selon la localisation du code source. Prenons l'exemple à la figure 16.

```
public class A
{
      public int a;
}

public class B extends A
{}

public class C
{
      public void m(B b)
      {     b.a = 2;    }
}
```

Figure 16 Exemple d'héritage.

Cela signifie que l'attribut « a » appartient à la classe A et non à la classe B. Donc, si on applique la règle d'impact du retrait d'attribut sur « A.a », le modèle MICJ va faire une erreur de prédiction. La règle d'impact spécifie que l'on va retirer les utilisations de l'attribut dans les classes qui héritent, localement et dans les classes associées. Donc, l'utilisation de l'attribut « a » dans la méthode « C.m() » ne sera pas prédit puisque la classe « C » est associée à la classe « B » et non pas à la classe « A ». Pourtant, il est évident que dans l'instruction « b.a = 2 » que l'on fait référence à l'attribut « a » que la classe « B » hérite de la classe « A ». On peut retrouver le même genre d'erreur sur les appels de méthodes héritées.

Cette erreur est due à une question d'interprétation du modèle objet qui n'est pas traitée et spécifiée par le modèle MICJ actuellement.

4.7.1.2 Ajout de classe

Dans le cas du modèle MICJ, ce qui diminue son « rappel », donc sa capacité à trouver les classes réellement impactées, c'est le changement à prédire « ajout d'une classe ». Autant pour le modèle MICJ que pour celui de Chaumun, il est impossible de prédire quel sera l'impact de ce changement (qui est en fait un ajout) sans faire intervenir l'utilisateur. Donc, quand on observe l'impact réel, ni l'un ni l'autre n'arrive à identifier les classes impactées (ce qui est un peu normal). La raison de cette incapacité est très simple. Elle est due au fait que les deux modèles cherchent des impacts dans des classes qui ont une quelconque relation avec la classe modifiée. Mais dans ce cas précis, l'ajout d'une classe, la classe modifiée n'a de lien avec aucune classe puisqu'inexistante à l'analyse. Pour pouvoir faire une prédiction, il faudrait connaître avec quelles classes elle établira des liens. Donc, connaître l'intention qui va avec l'ajout de la nouvelle classe. Et ça, aucun modèle ne le prend en compte.

Il serait néanmoins possible de déterminer l'impact de l'ajout d'une classe dans certaines conditions et en considérant certaines hypothèses comme vraies. Pour le comprendre, il faut se rappeler de l'idée de l'analyse d'impact de l'ajout d'une méthode à une classe. Certains de ses impacts (comme l'appel de cette nouvelle méthode) spécifient les classes associées. Il est possible, que suite à ce changement, que de nouvelles classes soient associées sur le principe que l'ajout d'une méthode (fonctionnalité nouvelle) pouvant être utile à de nouvelles classes. Malheureusement, prédire qu'elles seront ces nouvelles classes – s'il y en a – est simplement impossible sans connaître les intentions derrière ce changement tout comme pour l'ajout d'une classe. C'est pour cela que lorsqu'on spécifie les classes associées, on se restreint aux classes déjà associées à la classe où on effectue l'ajout de méthode. Dans le cas du changement lié à l'ajout d'une classe, les liens ne sont pas déjà préétablis avec cette classe. Par contre, pour la prédiction, si on peut spécifier de quelle classe héritera la nouvelle classe et quelle interface elle implémentera, on pourrait considérer les associations que possèdent ses classes « frères » (classes ayant le même héritage ou implémentation des mêmes interfaces), ou de la classe mère et de ses interfaces sous

le principe que des classes avec un lien de parenté devraient avoir un comportement similaire et apparenté.

En raison de ces hypothèses, cet aspect n'a pas été pris en considération. Par contre, offrir l'option dans l'implémentation pour l'utilisateur, pourrait être intéressant surtout si l'expérimentation peut montrer un taux acceptable de précision et de rappel avec les hypothèses.

4.7.1.3 Réduction d'instruction

Dans le pattern façade, on a détecté une erreur de prédiction pour les impacts certains. C'est l'unique erreur dans les prédictions d'impact avec certitude qui a été trouvée dans les différentes expérimentations effectuées. Elle est due à un phénomène de réduction d'instruction permis par le changement effectué ce qui fait que le modèle ne prédit pas le bon changement.

À la méthode « RegularScreen.newline( ) », un paramètre a été ajouté: un entier nommé « count ». Cette méthode permet simplement d'afficher un retour de chariot à la console. L'ajout du paramètre « count » permet d'ajouter un nombre « count » de retour de chariot. Dans la méthode « OutputFacade.printDecoration( ) », la méthode « RegularScreen.newline( ) » est appelée consécutivement à deux reprises. Le modèle MICJ prédisait que pour chacun de ces appels, il y aurait le changement de l'appel (ajout du passage d'un paramètre à l'appel). Mais ce paramètre, permettant de choisir le nombre de retour de chariot a fait en sorte que les deux appels en sont devenus qu'un seul avec « 2 » en paramètre. Il y a donc un des deux appels qui ne sera pas impacté comme le modèle MICJ le prédit. Il sera plutôt retiré.

Cette prédiction pourrait être corrigée par un utilisateur qui sait quelle sera l'utilité du paramètre et qui peut comprendre l'effet de réduction d'instructions. Par contre, au niveau du modèle, on ne peut pas prédire ce genre de détail lié à l'intention du changement.

4.7.1.4 Interférences entre changements et invisibilités des changements

Dans le pattern « memento », on effectue un changement de type de l'attribut « count » de la classe « Counter ». Il passe du type entier au type réel. Le modèle MICJ prévoit que les instructions utilisant l'attribut « count » pourraient être changées, cependant ça n'a pas eu lieu. Une de ses instructions est un appel de méthode de la classe « CounterMemento » en passant en paramètre l'attribut « count ». Si on ne change rien à la classe « CounterMemento », on serait obligé de faire un changement de type explicite (un « cast ») pour pouvoir passer l'attribut en paramètre et un des changements incertains aurait eu lieu. Mais cela n'est pas le cas car on a aussi changé l'attribut « state » de la classe « CounterMemento » d'entier à réel. De ce changement découle d'autres changements qui font en sorte que la méthode appelée avec « count » comme paramètre accepte le réel en paramètre. Cela fait en sorte qu'au niveau du code, on n'y voit aucun changement, mais qu'au niveau sémantique, il y en a un puisque le type du paramètre a changé au niveau de l'appel. Ce sont des changements invisibles à l'analyse syntaxique du code à cause de l'interférence de plus d'un changement au niveau du code.

Ce type d'interférence peut arriver régulièrement lorsqu'on effectue des changements dans le code. Au niveau prédictif, on ne peut pas le prévoir car ces changements invisibles dépendent des choix d'implémentation que fait le programmeur.

Dans le cas du pattern « memento », cela a diminué la précision des impacts incertains et c'est la post-analyse qui permet de comprendre le phénomène. Mais dans d'autre cas, comme dans le dernier groupe d'expérimentation, cela à un impact sur la prédiction des impacts certains.

Il est important de souligner ici ce qui a été considéré comme un impact réel dans les expérimentations. Pour qu'il y ait un impact réel, il devait y avoir un changement au niveau de l'écriture du code. Ceci sous entend qu'un changement de la sémantique du code n'a pas été considéré comme un impact. L'analyse a été faite ainsi pour diverses raisons. Premièrement, elle a été faite manuellement et à la lecture du code. Cela rend difficile une lecture parfaite de la sémantique et la détection de ce genre de changements entre deux versions. La seconde raison de cette technique d'analyse est

que l'un des objectifs du modèle MICJ est de prédire l'ampleur de l'impact. Donc, la quantité de changements que le programmeur devra apporter au code.

Le modèle MICJ est aussi un modèle dont les utilisations sont diversifiées. Une de ses possibilités est d'être utilisé lors des tests de régression. Lorsqu'on refait un test qui marchait et que suite à des changements il ne fonctionne plus, on va vouloir savoir quels changements ont affecté ce test.

4.7.2 Comparaisons générales

Le tableau XV a pour objectif de montrer les différences entre les deux modèles qui ont été précédemment présentées.

Le modèle de Chaumun est sujet à interprétation et comporte un certain nombre de lacunes. On peut, entre autres, souligner un certains nombre de changements n'ayant aucun impact selon le modèle de Chaumun tel que l'ajout d'un attribut, le changement de visibilité d'un attribut de protégée à publique, de privée à publique et de privée à protégée, le passage de non-statique à statique d'un attribut et l'ajout d'une classe. Il est à noter que dans la majorité des cas, les impacts sont facultatifs. Par exemple, l'ajout d'un attribut de classe n'oblige en rien de l'utiliser. Par contre, lorsqu'on fait un changement, on s'attend à ce que ça ait un objectif et ajouter un attribut de classe sans l'utiliser est inutile.

L'autre point faible du modèle de Chaumun au niveau de l'interprétation est son manque de précision (niveau général – classe). En indiquant uniquement quelles classes seront impactées, le modèle donne peu d'indices sur la taille réelle de l'impact. Selon le modèle, si un changement effectué dans la classe *A* à un impact sur la classe *B*, cela ne nous dit pas si on a un seul ou une grande quantité de changements à effectuer dans la classe *B*. De plus, en ne spécifiant pas la nature de l'impact, le modèle de Chaumun ne permet pas une analyse en cascade.

Tableau XV

Comparaison générale des modèles.

Critère	Modèle d'impact de changement en Java	Modèle de Chaumun pour Java
Nombre de changements structurels	41	52
Nombre de changements non-structurels	26	0
Nombre de règles d'impact	41	33
Niveau de précision de l'impact	Changement atomique, méthode et classe	Classe
Niveau de complexité des règles d'impact	Élevé	Basse
Niveau de complexité du modèle	Élevé	Moyen
Base de construction du modèle	Règle grammaticale et sémantique du langage Java	Modèle de Chaumun pour langage C++

Le modèle MICJ apporte une plus grande précision dans les impacts - quels sont les impacts et où on les trouve. Il est donc plus explicite et plus précis que celui de Chaumun, ce qui permettra éventuellement d'offrir un meilleur soutien dans les différentes tâches pouvant être associées à un changement donné et à la maintenance de façon globale.

4.7.3 Comparaison de l'impact en cascade

Avec le modèle de Chaumun, il est difficile d'analyser l'effet en cascade efficacement car le modèle ne cible pas un changement en particulier. Même si le modèle de Chaumun prédit correctement l'impact d'une classe, on n'aura toujours pas l'information sur la nature de l'impact. Donc, si l'impact était le changement de signature d'une méthode qui a un impact sur d'autres classes, il serait impossible avec le modèle de Chaumun d'obtenir cette information nécessaire pour pouvoir pousser l'analyse plus loin : faire de l'analyse en cascade.

Le modèle MICJ, en spécifiant clairement quel changement est attendu après un changement initial donné, permet de faire le lien entre les différents impacts et prédire l'effet en cascade.

CHAPITRE 5

PROTOTYPAGE

5.1 Introduction

Dans le but d'automatiser l'analyse avec le modèle MICJ, on a développé un prototype de représentation des règles. L'objectif étant d'obtenir un logiciel souple sur l'interprétation des règles d'impacts puisque les règles sont susceptibles d'être modifiées.

Dans ce contexte, une structure en XML permettant de générer les règles d'impacts a été définie. Un interpréteur XML a été développé. Il extrait les règles en tant qu'arbre XML. Cet arbre est une structure stable représentant les règles à partir desquelles on peut extraire facilement l'information nécessaire. Un interpréteur d'arbre XML permet de générer une structure objet des règles d'impacts extraites du code XML pouvant être utilisée pour appliquer concrètement les règles.

5.2 XML

XML (« **e**xtensible **m**arkup **l**anguage ») est un langage de type balise. Ce qui le démarque des autres langages de ce type est le fait qu'on n'est pas limité à certaines balises comme le langage HTML. Dans le prototype, on n'utilise que les notions de bases du XML. Ces notions sont présentées dans ce qui suit.

Chaque balise XML porte un nom et peut posséder un certain nombre d'attributs. De plus, à l'intérieur d'une balise, il peut être défini d'autres sous-balises. Deux balises de même nom et de même niveau sont considérées comme des balises de même type. Deux balises du même type ne sont pas obligées d'avoir les mêmes attributs et les mêmes sous-balises.

Un code XML peut être représenté par un arbre avec des nœuds. Chaque balise correspond à un nœud de l'arbre. Un exemple est donné à la figure 17 d'un code XML et de son arbre.

C'est à partir de cet arbre que les règles d'impacts seront reconstituées et importées dans le prototype.

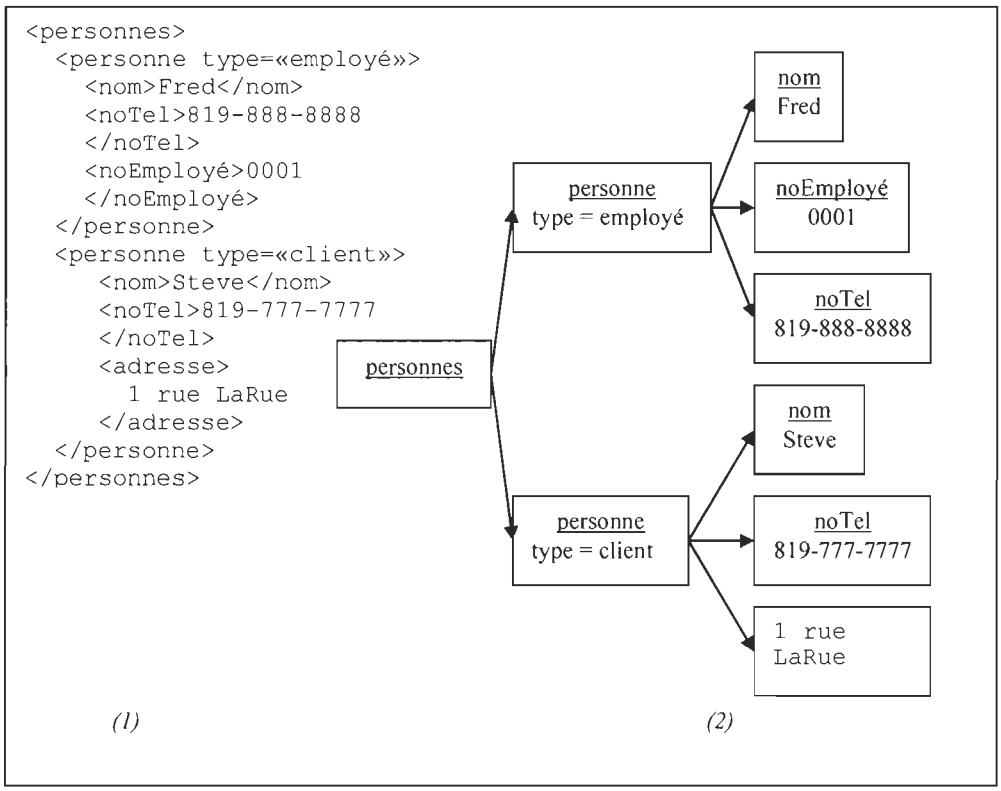

Figure 17 Code source XML (1) et arbre XML correspondant (2).

5.3 Règles d'impacts sous forme XML

Pour mettre les règles d'impacts sous forme XML et conserver l'avantage de souplesse du XML, les règles d'impacts pour chaque changement ont été séparées de la description des changements, des cibles et des relations entre classes. Il y a, donc, plusieurs nœuds racine dans l'arbre : « changementModel », « cibleModel », « scopeModel » et « text ».

L'arbre « cibleModel » contient la liste des cibles du modèle. Chaque fois qu'une règle d'impact fait référence à une cible ou à une condition, elle doit se référer à « cibleModel ». L'arbre est constitué de balises « cibleType » ayant un attribut « id » qui est l'identifiant unique du type de la cible. Cet identifiant est utilisé par les règles d'impact pour référencer une cible.

La figure 18 donne un aperçu du code XML correspondant à l'arbre « cibleModel » et l'arbre qui lui est associé.

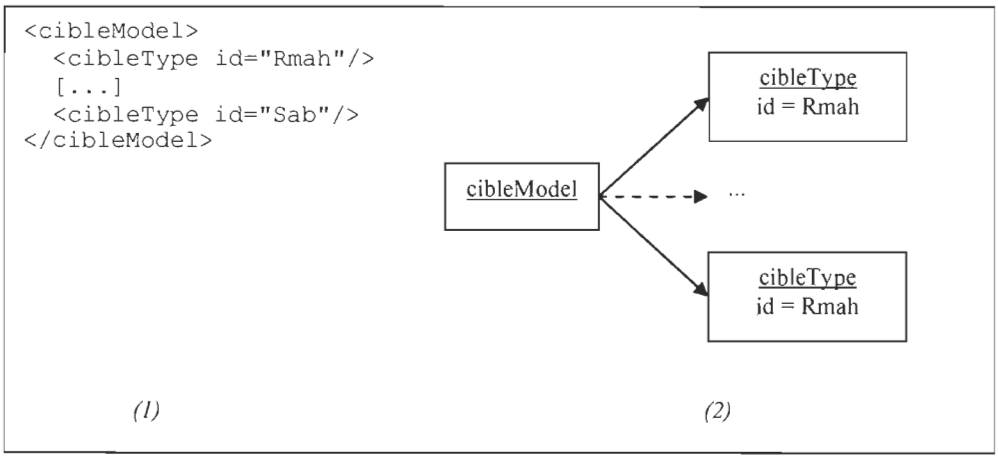

Figure 18 Code source XML (1) et arbre XML correspondant (2).

L'arbre « scopeModel » contient la liste des relations entre classes du modèle. Chaque fois qu'une règle d'impact fait référence à une relation entre classes, il doit se référer à « scopeModel ». L'arbre est constitué de balises « scopeType » ayant un attribut « id » identifiant unique du type de la relation. Cet identifiant est utilisé par les règles d'impact pour référencer une relation.

La figure 19 donne le code XML correspondant à l'arbre « scopeModel » et l'arbre qui lui est associé.

L'arbre « text » a été créé pour atteindre deux objectifs. Premièrement, contenir les informations textuelles des divers éléments des règles d'impacts. C'est-à-dire, les diminutifs, le nom complet et la description de chaque changement atomique venant de « changementModel », de chaque relation entre classes venant de « scopeModel » et de chaque cible venant de « cibleModel ». Cela permet d'alléger le code XML lié aux règles d'impacts en séparant ainsi forme et fond. Le deuxième objectif est de permettre éventuellement l'extension du code XML vers une version multilingue. La transition est simple, mais elle sera expliquée après avoir détaillé l'arbre « text ».

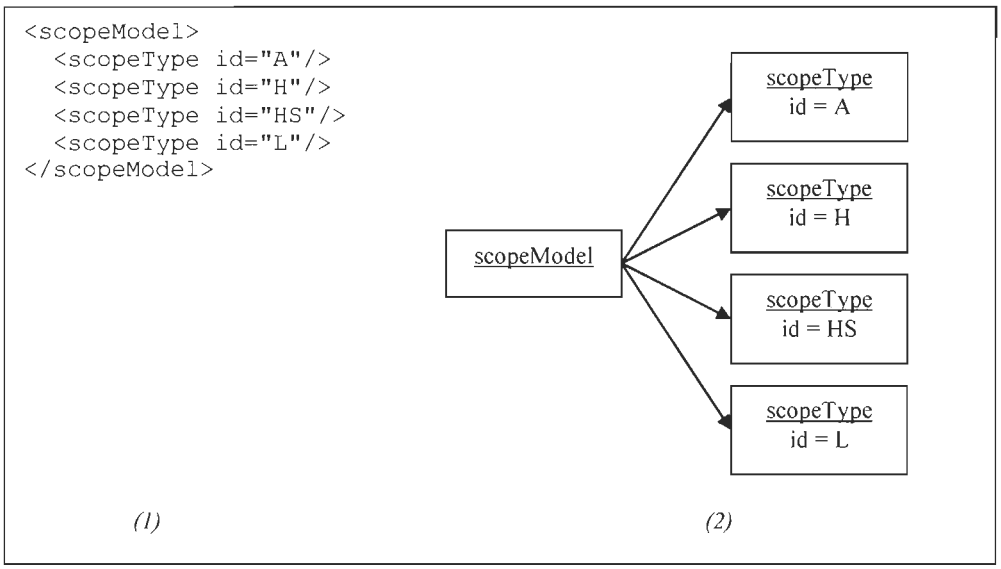

Figure 19 Code source XML (1) et arbre XML correspondant (2).

L'arbre « text » contient donc les informations texte correspondants aux différents éléments du modèle. À chaque identifiant dans l'arbre « changementModel », « scopeModel » et « cibleModel » correspond une balise dans l'arbre « text ». L'arbre est constitué de trois balises. La première, « changement », possède les balises correspondantes aux identifiants de l'arbre « changementModel ». La deuxième, « scope », possède les balises correspondantes aux identifiants de l'arbre « scopeModel ». La troisième, « cible », possède les balises correspondantes aux identifiants de l'arbre « cibleType ». Les balises correspondantes aux identifiants des différents arbres possèdent l'attribut « name » et « description ». L'attribut « name » est le diminutif de l'élément et l'attribut « description » est le nom complet de l'élément.

La figure 20 donne un extrait du code XML correspondant à l'arbre « text » et l'arbre qui lui est associé.

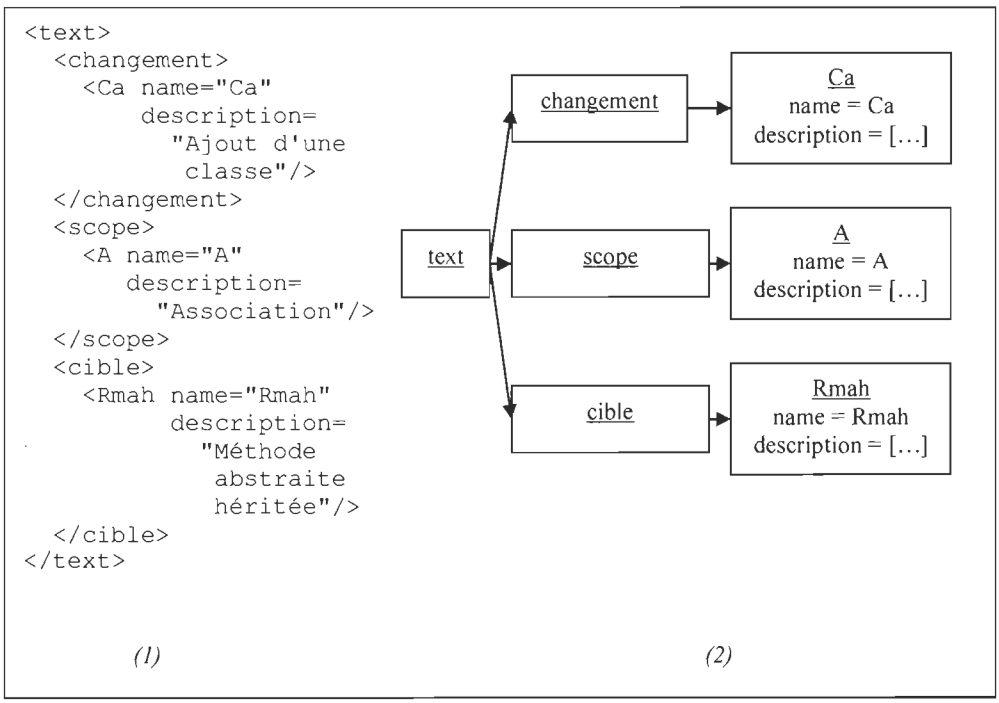

Figure 20 Code source XML (1) et arbre XML correspondant (2).

L'arbre « changementModel » contient la liste des changements atomiques du modèle ainsi que leur règle d'impacts. Le nœud racine « changementModel » est composé de balise « changementType » qui a un attribut « id » qui est l'identifiant du changement. Comme pour les autres arbres, cet identifiant permet de faire la liaison entre le changement, ses informations textuelles et sa présence en tant qu'impact dans les règles d'impacts. Chaque nœud « changementType » est composé d'une liste de balises « impact » représentant chaque impact de la règle d'impact de chaque changement. Un nœud « impact » a comme attribut « type » l'identifiant d'un changement atomique de l'impact, « or » et « and » sont les identifiants d'un changement atomique alternatif s'il y en a un et « obligation » qui est soit vrai soit faux représentant l'impact qui peut être certain ou incertain. Chaque nœud « impact » peut être composé des balises « scope », « cible » et « condition » représentant respectivement les relations entre classes, les cibles et les conditions de certitude de l'impact.

La figure 21 donne le code XML d'un nœud « changementType » et de l'arbre correspondant.

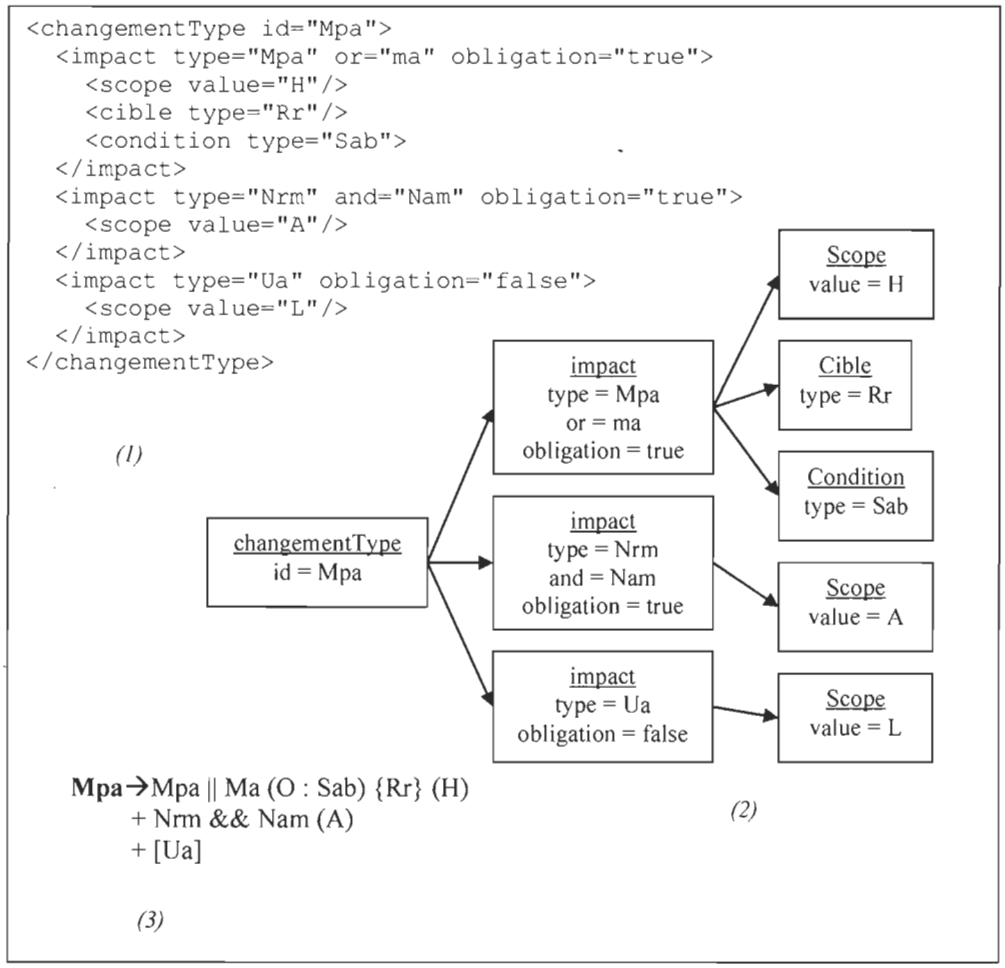

Figure 21 Code source XML (1), arbre XML correspondant (2) et règle d'impact correspondante.

5.4 Parseur XML

Le prototype comprend un parseur XML spécialisé pour le modèle écrit dans le langage Java. Il peut cependant être utilisé à d'autres fins.

Le parseur permet d'analyser les balises ouvertes avec leurs attributs, les balises fermées avec leurs attributs et permet aussi les commentaires. Ces derniers sont simplement retirés pour la formation de l'arbre XML. Une balise ouverte est une

balise contenant des sous-balises. Cela implique qu'elle possède une balise fermante. Une balise fermée est une balise ne contenant pas de sous-balises. Si une règle d'écriture n'est pas respectée ou connue par le parseur, celui-ci signalera l'élément d'erreur et le message d'erreur correspondant et arrêtera la construction de l'arbre immédiatement sans toutefois le détruire.

L'analyse se fait en plusieurs étapes et chaque classe du prototype à sa responsabilité. Dans un premier temps, les classes « Token.java » et « TokenParser.java » travaillent ensemble pour analyser le code XML et faire une liste de jetons représentant le code. Chaque jeton représente un élément reconnaissable du code : l'ouverture d'une balise (« < »), un identifiant, le égal (« = »), etc. Ils éliminent tous les commentaires lors de cette analyse. Par la suite, c'est la classe « XMLReader.java » qui s'occupe de lire la liste de jetons et de reconnaître les balises. C'est aussi cette classe qui vérifie si le code XML est valide et qui forme l'arbre au fur et à mesure que la lecture des jetons progresse.

La classe « XMLReader.java » crée un arbre « XMLTree.java ». Elle implémente la structure d'arbre dont les nœuds sont des objets de la classe « XMLNode.java ». Elle implémente aussi les fonctionnalités de parcours de l'arbre et de troncature de l'arbre qui seront utiles à l'analyse de l'arbre.

Chaque nœud de l'arbre possède sa liste d'objets de la classe « XMLAttribut.java » correspondant aux attributs des balises. Les nœuds possèdent aussi un objet générique de la classe « XMLObject.java ». Ceux-ci ne sont pas utilisés lors de la formation de l'arbre. Leur utilisation sera expliquée ultérieurement.

Un objet de la classe « XMLAttribut » contient un attribut référence sur le nœud courant. C'est toujours à partir de celui-ci que l'on travaille. Évidemment, il est possible de parcourir manuellement l'arbre, mais en allant vers les feuilles seulement. On ne peut pas retourner directement au nœud parent. Ce sont les opérations de recherche et de troncature de la classe « XMLTree.java » qui font presque tout le travail pour la reconstitution du MICJ. La première fonction de troncature est celle implémentée par la méthode « getXMLTreeFromCurrent ». Elle

retourne simplement un arbre ayant pour racine le nœud courant. La seconde méthode de troncature est « getInstanceOf » qui prend en paramètre un chemin. Chaque mot du chemin, divisé par un point, correspond au nom d'un nœud. Cette méthode retourne un arbre formé des nœuds correspondant au chemin. La dernière méthode de troncature est « lookFor ». On doit lui fournir un chemin, le nom d'un attribut et la valeur de cet attribut. Selon le même principe que la méthode « getInstanceOf », un arbre avec tous les nœuds est retourné. Ces noeuds correspondent au chemin et ayant un attribut du nom et de la valeur mentionnée.

La figure 22 montre le schéma du fonctionnement du parseur.

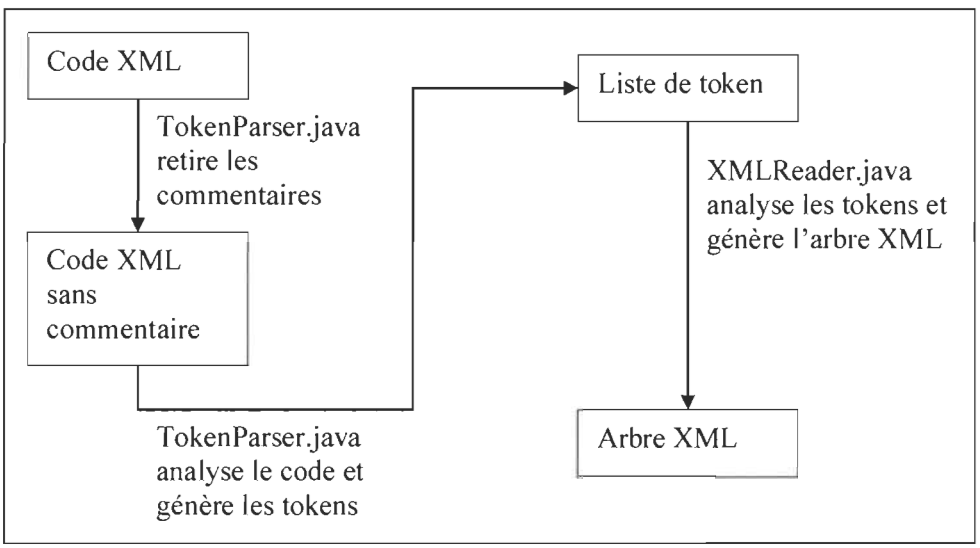

Figure 22 Schéma de la génération de l'arbre XML.

5.5 Modèle objet du modèle MICJ

Le modèle objet du MICJ a été implémenté de façon simple. C'est-à-dire qu'à chaque élément du modèle correspond un objet ayant pour attribut les objets qui lui sont rattachés. Le modèle objet est aussi très près du modèle XML. La figure 23 présente le diagramme de classes de l'implémentation du modèle.

Comme le montre le modèle, la classe « ChangementType.java » est l'objet représentant chacun des changements atomiques. Ils ont donc leur identifiant ainsi qu'une règle d'impact représentée par une liste d'objets de la classe « Impact.java »

ou « ImpactOrAnd.java » pour les impacts ayant une alternative « ET » ou « OU ». Chaque impact possède ses attributs « Cible.java » pour les cibles et « Scope.java » pour les relations entre classes. Les types de cibles et de relations entre classes, représentés par « CibleType.java » et « ScopeType.java », ont été séparés des impacts pour faciliter l'éventuelle implémentation de l'analyse d'impact. Cela fait en sorte que chaque impact possède sa propre instanciation de l'objet « Cible.java » ou « Scope.java », mais ceux-ci, font tous référence à des instanciations uniques de leur type. Cela permet de séparer la représentation de l'implémentation des fonctionnalités associées à l'analyse.

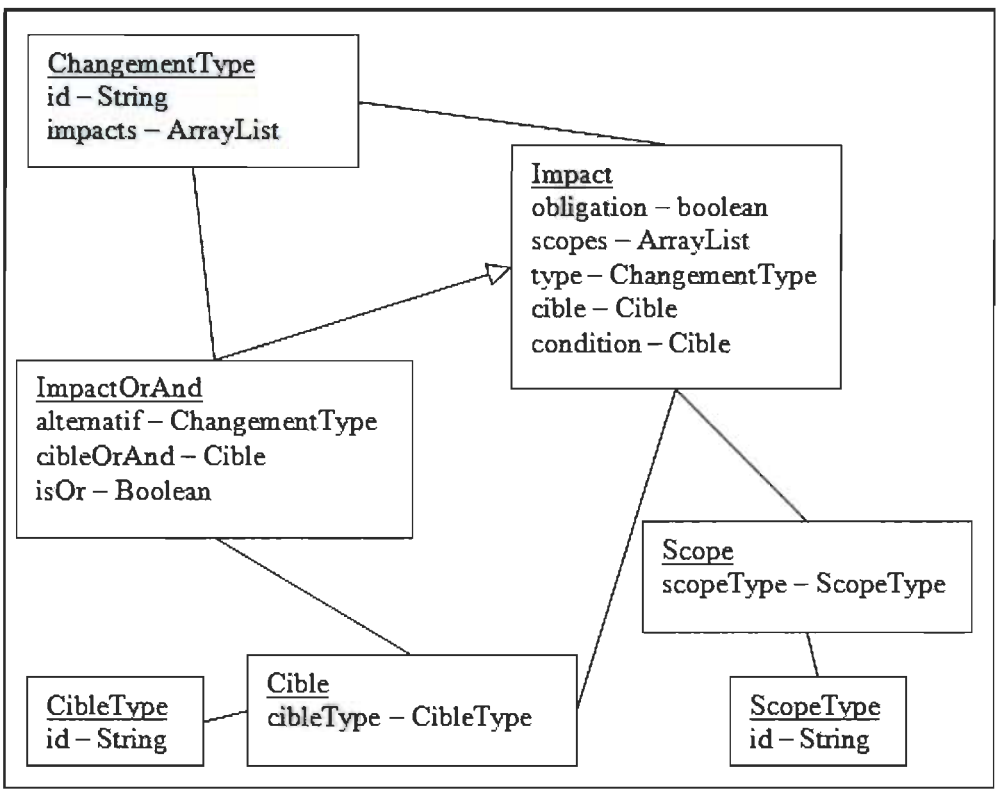

Figure 23 Diagramme de classes de l'implémentation du modèle.

Pour parvenir à obtenir le modèle objet du modèle MICJ à partir du code XML, la classe « ModelExtractor.java » instancie un objet « XMLTree.java » qui analyse et forme l'arbre XML du modèle. Ensuite, plusieurs arbres sont extraits : l'arbre contenant les types de changements, l'arbre contenant les types de relations entre

classes, l'arbre contenant les types de cibles et les trois arbres correspondants aux descriptions textuelles de ceux-ci.

L'extracteur de modèle fonctionne ensuite en deux étapes. En premier lieu, il va créer les objets de type ; les objets des classes « ChangementType.java », « ScopeType.java » et « CibleType.java ». On associe ces objets aux nœuds « XMLNode.java » grâce à l'attribut « object » de la classe « XMLObject.java ». Ce dernier agit uniquement comme conteneur pour la formation du modèle objet du MICJ. Cela permet de récupérer facilement les objets de types des classes « ChangementType.java », « ScopeType.java » et « CibleType.java » à travers l'arbre XML qui implémente les fonctionnalités de recherche.

La deuxième étape est bien sûr celle de la formation des règles d'impact. L'extracteur regarde chacun des nœuds « changementType » pour en extraire chaque impact. Ceux-ci seront analysés et les objets de classe « Impact.java », « ImpactOrAnd.java », « Cible.java » et « Scope.java » sont instanciés et associés aux instanciations uniques des objets « ChangementType.java », « CibleType.java » et « ScopeType.java » qui sont retrouvés dans les arbres XML puisque déjà présents dans les nœuds sous l'attribut « object » de classe « XMLObject.java ».

5.6 Utilisation du modèle objet du modèle MICJ

La représentation objet du MICJ peut être étendue dans ses fonctionnalités pour permettre une analyse d'impact automatisée. Elle est surtout nécessaire pour l'utilisation du modèle MICJ sur des programmes de moyenne et grande envergure. En effet, dans les expérimentations faites sur des programmes de petites tailles, l'analyse d'impact réalisée manuellement d'un seul changement pouvait prendre plusieurs minutes (sans analyse d'impact en cascade), malgré la maîtrise du modèle MICJ et la connaissance préalable du fonctionnement du programme. Ces derniers éléments ne sont pas toujours possibles sur des programmes plus volumineux. Pour être réellement pratique et réduire les coûts relatifs au temps d'utilisation, le modèle MICJ doit être automatisé.

CHAPITRE 6

CONCLUSIONS

La revue de la littérature a montré que plusieurs approches reliées à l'analyse d'impact ont été proposées. Toutes ces approches possèdent des avantages et des inconvénients. Cette revue a aussi montré que peu de modèles (réellement utilisables) sont disponibles pour supporter l'analyse d'impact soit à partir du modèle objet des applications ou bien à partir de leur code source. Le plus reconnu et le plus élaboré est le modèle de Chaumun [9]. Il a été développé à l'origine pour le langage C++ et ensuite adapté pour le langage Java par Kabaili et al.[1][3][4][5]. Néanmoins, ce modèle à granularité forte offre une analyse très générale de l'impact d'un changement. De plus, il ne permet pas d'apporter des informations précises aux programmeurs sur les impacts dans le code.

Le modèle proposé dans ce mémoire supporte une analyse prédictive. Il est plus précis et offre la possibilité de faire de l'analyse d'impact en cascade, ce qui est très utile dans ce genre de contexte. Il est aussi capable de fournir le plus d'informations possibles pour aider à l'application du changement.

Grâce aux différentes évaluations effectuées, il est possible de dire que le modèle MICJ parvient à obtenir des résultats très intéressants en termes de précision et de rappel. De plus, il est possible, grâce à ses règles, de faire de l'analyse en cascade. Cela permet d'avoir une analyse d'impact du changement en profondeur en plus de fournir de l'information au programmeur qui désire effectuer un changement donné.

Les nombreuses expérimentations effectuées ont aussi permis de faire ressortir quelques limites du modèle proposé. Ces quelques lacunes, discutées dans le mémoire, sont essentiellement dues à des éléments liés plus aux intentions des changements, ce qui les rend difficiles à cerner (et donc à corriger) puisque le modèle MICJ n'est en fait pas prévu pour cela.

L'implémentation du modèle devra être complétée pour étendre son évaluation sur des programmes de plus grande taille. Par ailleurs, et dans cette optique, une piste intéressante réside dans la possibilité d'intégrer un certain « apprentissage » dans l'utilisation du modèle (par le biais entre autre de l'analyseur). Il serait alors possible, grâce à la pré et post analyse d'extraire des données permettant d'avoir des statistiques d'occurrence des impacts pour un changement et d'occurrence des changements. Cela permettrait de mitiger l'importance de l'impact de chaque changement dans l'analyse prédictive permettant ainsi de mieux estimer les coûts et les efforts liés à un changement. Le modèle pourrait aussi être adapté pour d'autres langages tel que AspectJ qui est une extension du langage Java.

BIBLIOGRAPHIE

[1] M.K Abdi, H. Lounis et H. Sahraoui, *Using coupling metrics for change impact analysis in ojbect-oriented systems*, 10th ECOOP Workshop on quantitive approaches in object-oriented software engineering, 2006.

[2] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller, François Lustman et Guy Saint-Denis, *Desing properties and object-oriented software changeability*, CSMR '00, Proceedings of the Conference on Software Maintenance and Reengineering, 2000.

[3] Hind Kabaili, Rudolf K. Keller et François Lustman, *A change impact model encompassing ripple effect and regression testing*, In Proceedings of the Fifth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering, 2001.

[4] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller et François Lustman, *A change impact model for changeability assessment in object-oriented software systems*, CSMR '99, Proceedings of the Third European Conference on Software Maintenance and Reengineering, 1999.

[5] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller et François Lustman, *A change impact model for changeability assessment in object-oriented software systems*, Journal of Science of Computer Programming - Software maintenance and reengineering (CSMR 99) archive, Volume 45 Issue 2-3, 2002.

[6] Xiaoxia Ren, Fenil Shah, Frank Tip, B. G. Ryder et Ophelia Chesley, *Chianti:A tool for change Impact analysis of Java Programs*, OOPSLA '04, Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004

[7] M.K Abdi, H. Lounis et H.Sahraoui, *Analyse et prédiction de l'impact de changements dans un système à objets : Approches probabiliste*, In proc. of LMO 2009, Nancy, France, 2009.

[8] M.K Abdi, H. Lounis et H. Sahraoui, *Analyzing Change Impact in Object-Oriented Systems*, In proceedings of the 32nd EUROMICRO Software Engineering and Advanced Applications Conference, Cavtat/Dubrovnik (Croatia), 2006.

[9] M. A. Chaumun, *Change Impact Analysis in Object-Oriented Systems: Conceptual Model and Application to C++*, Master's thesis, Université de Montréal, Canada, November 1998.

[10] Sumita Das, Wayne G. Lutters et Carolyn B. Seaman, *Understanding Documentation Value in Software Maintenance*, CHIMIT '07, Proceedings of the 2007 Symposium on computer human interaction for the management of information technology, 2007.

[11] Pieter Hooimeijer et Westley Weimer, *Modeling Bug Report Quality*, Proceedings of the twenty-second IEEE/ACM International conference on Automated Software Engineering (ASE '07), pp. 34–43, Atlanta, Georgia, 2007.

[12] Keith Bennett et Vaclav Rajlich, *Software Maintenance and Evolution : A Roadmap*, The Future of Software Engineering, ACM Press, 73 – 87, 2000.

[13] Andrea De Lucia, Eugenio Pompella et Silvio Stefanucci, *Effort estimation for corrective Software Maintenance*, SEKE '02, Proceedings of the 14th International conference on software engineering and knowledge engineering, 2002.

[14] Chris F. Kemerer, *A Longitudinal Analysis of Software Maintenance Patterns*, ICIS '97, Proceedings of the eighteenth International Conference on Information Systems, 1997.

[15] Yogesh Singh et Bindu Goel, *A Step Towards Software Preventive Maintenance*, ACM SIGSOFT Software Engineering Notes, Volume 32 Issue 4, 2007.

[16] Pankaj Bhatt, Gautam Shroff et Arun K. Misra, *Dynamics of Software Maintenance*, ACM SIGSOFT Software Engineering Notes, Volume 29 Issue 5, 2004.

[17] Sergio Cozzetti B. de Souza, Nicolas Anquetil et Káthia M. de Oliveira, *A Study of the Documentation Essential to Software Maintenance*, SIGDOC '05, Proceedings of the 23rd Annual International Conference on design of communication: documenting & designing for pervasive information, 2005.

[18] Ian Sommerville, *Software Engineering*, Seventh edition, Pearson, Addison Wesley, 2004.

[19] Daniel St-Yves, *Dépendances et gestion des modifications dans les systèmes orientés objet : utilisation des graphes de contrôle*, Thèse de maîtrise, Université du Québec à Trois-Rivières, Canada, décembre 2007.

[20] S. Barros, Th. Bodhun, A. Escudie, J.P. Voidrot, *Supporting Impact Analysis : A semi automated technique and associated tool*. Proc. of the

1995 IEEE Conf. on Software Maintenance, pp. 42-51, Piscataway, NJ, 1995.

[21] Badri, L., Badri, M., and St-Yves, D. 2005. *Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique*. In Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05) - Volume 00, IEEE Computer Society, 2005.

[22] L. Badri, M. Badri & Daniel St-Yves, *Analyse de l'impact des changements: Une étude expérimentale sur deux approches statiques*, in Revue Génie Logiciel, Numéro 82, Paris, Septembre 2007.

[23] L. Badri, M. Badri & Daniel St-Yves, *Static Impact Analysis for Object-Oriented Programs: An experimental comparison on reduced-call-path and slicing based approaches*, in Proceedings of the 20th International Conference on Software and Systems Engineering and their Applications, Paris, Décembre 2007.

[24] J. Law, G. Rothermel, *Whole Program Path-Based Dynamic Impact Analysis*. Proc. of the International Conf. on Software Engineering, pp. 308-318, 2003.

[25] Hassan, A.E.; Holt, R.C., *Predicting change propagation in software systems*, Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on , vol., no.pp. 284- 293, 11-14, Sept. 2004

[26] S. Barros, Th. Bodhun, A. Escudie, J.P. Voidrot, *Supporting Impact Analysis : A semi automated technique and associated tool*. Proc. of the 1995 IEEE Conf. on Software Maintenance, pp. 42-51, Piscataway, NJ, 1995.

[27] A. Orso, T. Apiwattanapong, and M.J. Harrold, *Leveraging field data for impact analysis and regression testing*. Proc. of European Software Engineering Conf. And ACM SIGSOFT Symp. on the foundations of software engineering (ESEC/FSE'03), Helsinki, Finland, Sept. 2003.

[28] A. Orso, T. Apiwattanapong, J.Law, G. Rothermel, and M.J. Harrold, *An Empirical Comparison of Dynamic Impact Analysis Algorithms*. Proc. of the International Conf. on Software Engineering (ICSE'04), , pp. 491-500, Edinburg, Scotland, 2004.

[29] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller and F. Lustman, *A change impact model for changeability assessment in object-oriented software systems*, Science of Computer Programming, Volume 45, Issues 2-3, , November-December 2002, Pages 155-174.

[30] Barbara G. Ryder and Frank Tip, *Change Impact Analysis for object-Oriented Programs*. In ACM SIGPLAN-SIGSOFT Worshop on Program Analysis for Software Tools and Engineering, pages 46-53. ACM Press, 2001.

[31] Yau, S.S.; Collofello, J.S.; MacGregor, T., *Ripple effect analysis of software maintenance*, Computer Software and Applications Conference, COMPSAC '78, 1978

[32] Yamin Wang and Wei-Tek Tsai and Xiaoping Chen and Sanjai Rayadurgam 1996. *The Role of Program Slicing in Ripple Effect Analysis*. Proc. of SEKE, 1996.

[33] Black, S. 2001. *Computing ripple effect for software maintenance*. Journal of Software Maintenance 13, 4, Sep. 2001.

[34] Louise Martin, *Statistique avec applications aux sciences du loisir, de la culture, du tourisme et des communications – Traitement de données avec Microsoft Excel 2000*, Trois-Rivières : Les éditions SMG, 2001.

[35] Gérald Baillargeon, *Méthodes statistiques volume 2 – Méthodes d'analyse de régression linéaire simple et de régression multiple, analyse de corrélation linéaire simple*, Trois-Rivières : Les éditions SMG, 1995.

[36] Joseph Lee Rodgers et W. Alan Nicewander, Thriteen ways to look at the correlation coeficient, *The American Statistician*, vol. 42, no. 1, Feb. 1988.

[37] Hans Van Vliet, *Software Engineering Principles and Practices*, second edition, Wiley, 2000.

[38] April-Abran, *Améliorer la maintenance du logiciel*, édition Loze-Dion, 2006.

[39] Alain April, Alain Abran, Reiner R. Dumke, *SM^{CMM} Model to Evaluate and Improve the Quality of the Software Maintance Precess*, 8th European Conference on Software Maintenance and Reengineering, 2004.

ANNEXE A

Modèle de Chaumun

A.1 Calcul de l'impact de Chaumun

Cette exemple est tiré de [9] et à été conservé tel qu'écrit dans le document. La partie des exemples dans du code à été retirée.

Table A.17 – m.2.8.2 : Method deletion

Which Change?	*Links of class C2 with class C1*					**Result?**
Change on class C1	**S**	**G**	**H**	**I**	**F**	*Impact on class C2*
Virtual method & Non virtual method	Y	Y	Y	Y	Y	**I**
	Y	Y	Y	Y	N	**I**
	Y	Y	Y	N	Y	**X**
	Y	Y	Y	N	N	**X**
	Y	Y	N	Y	Y	**I**
	Y	Y	N	Y	N	**I**
	Y	Y	N	N	Y	**X**
	Y	Y	N	N	N	**X**
	Y	N	Y	Y	Y	**I**
	Y	N	Y	Y	N	**I**
	Y	N	Y	N	Y	**X**
	Y	N	Y	N	N	**X**
	Y	N	N	Y	Y	**I**
	Y	N	N	Y	N	**I**
	Y	N	N	N	Y	**X**
	Y	N	N	N	N	**X**
	N	Y	Y	Y	Y	**I**
	N	Y	Y	Y	N	**I**
	N	Y	Y	N	Y	**X**
	N	Y	Y	N	N	**X**
	N	Y	N	Y	Y	**I**
	N	Y	N	Y	N	**I**
	N	Y	N	N	Y	**X**
	N	Y	N	N	N	**X**
	N	N	Y	Y	Y	**I**
	N	N	Y	Y	N	**I**
	N	N	Y	N	Y	**X**
	N	N	Y	N	N	**X**
	N	N	N	Y	Y	**I**
	N	N	N	Y	N	**I**
	N	N	N	N	Y	**X**
	N	N	N	N	N	**X**

Canonical Expression :

(SGHIF' + SG'HIF' + S'GHIF' + S'G'HIF')

Reduced Expression :

I + *ie*(3.I.1)

All classes invoking the deleted method will be impacted. However, in the case of virtual methods, there are instances where the call may still be valid – see example below. There is also local impact.

The deleted method could have provided the definition for an inherited pure virtual method. After deletion, the pure virtual method will be inherited turning the class into an abstract one (assuming it were not abstract). Refer to *change in class from non-abstract to abstract (c.3.1.1)*.

A.2 Tableau des règles d'impacts du modèle de Chaumun pour le langage C++ [5]

Change Id	Change description	Impact expression	Local impact
v.1.1	Variable value change	—	—
v.1.2	Variable type change	S	L
v.1.3	Variable addition	—	—
v.1.4	Variable deletion	S	L
v.1.5	Variable scope change		
v.1.5.1	Public -> Private	S~F	—
v.1.5.2	Public -> Protected	S~H~F	—
v.1.5.3	Protected -> Private	SH~F	—
v.1.5.4	Protected -> Public	—	—
v.1.5.5	Private -> Public	—	—
v.1.5.6	Private -> Protected	—	—
v.1.6	Variable change (*Static/Non-static*)		
v.1.6.1	Static -> Non-static	S	L
v.1.6.2	Non-static -> Static	—	—
m.2.1	Method change (*Static/Non-static*)		
m.2.1.1	Static -> Non-static	I	L
m.2.1.2	Non-static -> Static	—	L
m.2.2	Method change (*Virtual/Non-Virtual/Pure virtual*)		
m.2.2.1	Virtual -> Non-virtual	—	—
m.2.2.2	Non-virtual -> Virtual	—	—
m.2.2.3	Virtual -> Pure virtual	H+*ie*(3.1.1)	L
m.2.2.4	Non-virtual -> Pure virtual	H+*ie*(3.1.1)	L
m.2.2.5	Pure virtual -> Virtual	H+*ie*(3.1.2)	L
m.2.2.6	Pure virtual - Non-virtual	H+*ie*(3.1.2)	L
m.2.3	Method return type change		
m.2.3.1	Non pure virtual method	I+*ie*(3.1.2)	L
m.2.3.2	Pure virtual method	H	L
m.2.4	Method implementation change	—	L
m.2.5	Method signature change		
m.2.5.1	Non pure virtual method	I	L
m.2.5.2	Pure virtual method	H	L
m.2.6	Method scope change		
m.2.6.1	Public -> Private		
m.2.6.1.1	Non virtual method	I~F	—
m.2.6.1.2	Virtual method	I~F	—
m.2.6.1.3	Pure virtual method	—	—
m.2.6.2	Public -> Protected		
m.2.6.2.1	Non virtual method	~HI~F	—
m.2.6.2.2	Virtual method	~HI~F	—
m.2.6.2.3	Pure virtual method	—	—
m.2.6.3	Protected -> Private		
m.2.6.3.1	Non virtual method	HI~F	—
m.2.6.3.2	Virtual method	HI~F	—
m.2.6.3.3	Pure virtual method	—	—
m.2.6.4	Protected -> Public		

A.2 Tableau des règles d'impacts du modèle de Chaumun pour le langage C++(Suite)

Change Id	Change description	Impact expression	Local impact
m.2.6.4.1	Non virtual method	—	—
m.2.6.4.2	Virtual method	—	—
m.2.6.4.3	Pure virtual method	—	—
m.2.6.5	Private -> Public		
m.2.6.5.1	Non virtual method	—	—
m.2.6.5.2	Virtual method	—	—
m.2.6.5.3	Pure virtual method	—	—
m.2.6.6	Private -> Protected		
m.2.6.6.1	Non virtual method	—	—
m.2.6.6.2	Virtual method	—	—
m.2.6.6.3	Pure virtual method	—	—
m.2.7	Method addition		
m.2.7.1	Pure virtual method	*ie*(3.1.1)	—
m.2.7.2	Virtual & non-virtal method	I+*ie*(3.1.2)	L
m.2.8	Method deletion		
m.2.8.1	Pure virtual method	*ie*(3.1.2)	—
m.2.8.2	Virtual & non-virtual method	I+*ie*(3.1.1)	L
c.3.1	Class change (*Abstract/Non-abstract*)		
c.3.1.1	Non-abstract -> Abstract	G+H+I	L
c.3.1.2	Abstract -> Non-abstract	H	L
c.3.2	Class friendship relation change		
c.3.2.1	Add friend	—	—
c.3.2.2	Delete friend	F(S+G+H+I)	—
c.3.3	Class deletion		
c.3.3.1	Non-abstract class	S+G+H+I	—
c.3.3.2	Abstract class	S+H+I	—
c.3.4	Class inheritance derivation		
c.3.4.1	Public -> Private	~F(S + I)	—
c.3.4.2	Public -> Protected	~H~F(S + I)	—
c.3.4.3	Protected -> Private	H~F (S + ~SG + ~SI)	—
c.3.4.4	Protected -> Public	—	—
c.3.4.5	Private -> Public	—	—
c.3.4.6	Private -> Protected	—	—
c.3.5	Class inheritance (*Virtual/Non-virtual*)		
c.3.5.1	Virtual -> Non-virtual	— L	
c.3.5.2	Non-virtual -> Virtual	— L	
c.3.6	Class addition	—	—
c.3.7	Class inheritance structure		
c.3.7.1	Add abstract class	S+G+H+I+ L *ie*(3.1.1)	
c.3.7.2	Add non-abstract class	H L	
c.3.7.3	Delete abstract class	H + F + *ie*(3.1.2)	L
c.3.7.4	Delete non-abstract class	H+F L	

A.3 Tableau des règles d'impacts du modèle de Chaumun pour le langage Java

Change Id	Change Description	Impact Expression
v.1.1	Variable value change	-
v.1.2	Variable type change	S+L
v.1.3	Variable addition	-
v.1.4	Variable deletion	S+L
v.1.5	Variable scope change	
v.1.5.1	Public → Private	S
v.1.5.2	Public → Protected	S~H
v.1.5.3	Protected → Private	SH
v.1.5.4	Protected → Public	-
v.1.5.5	Private → Public	-
v.1.5.6	Private → Protected	-
v.1.6	Variable change (Static/Non-static)	
v.1.6.1	Static → Non-static	S+L
v.1.6.2	Non-static → Static	-
m.2.1	Method change (Static/Non-static)	
m.2.1.1	Static → Non-static	I+L
m.2.1.2	Non-static → Static	L
m.2.2	Method change (Abstract/Non-abstract)	
m.2.2.1	Abstract → Non-abstract	H+ie(3.1.2)+L
m.2.2.2	Non-abstract → Abstract	H+ie(3.1.1)+L
m.2.3	Method Return type change	
m.2.3.1	Non-abstract method	H+ie(3.1.2)+L
m.2.3.2	Abstract method	H+L
m.2.4	Method implementation change	L
m.2.5	Method signature change	
m.2.5.1	Non-abstract method	I+ie(3.1.2)+L
m.2.5.2	Abstract method	H+L
m.2.6	Method Scope change	
m.2.6.1	Public → Private	
m.2.6.1.1	Non-abstract method	I
m.2.6.1.2	Abstract method	-
m.2.6.2	Public → Protected	
m.2.6.2.1	Non-abstract method	~H I
m.2.6.2.2	Abstract method	-
m.2.6.3	Protected → Private	
m.2.6.3.1	Non-abstract method	H I
m.2.6.3.2	Abstract method	-
m.2.6.4	Protected → Public	
m.2.6.4.1	Non-abstract method	-
m.2.6.4.2	Abstract method	-
m.2.6.5	Private → Public	
m.2.6.5.1	Non-abstract method	-
m.2.6.5.2	Abstract method	
m.2.6.6	Private → Protected	
m.2.6.6.1	Non-abstract method	-
m.2.6.6.2	Abstract method	-
m.2.7	Method addition	
m.2.7.1	Abstract method	ie(3.1.1)
m.2.7.2	Non-abstract method	I+ie(3.1.2)+L
m.2.8	Method deletion	
m.2.8.1	Abstract method	ie(3.1.2)
m.2.8.2	Non-abstract method	I + ie(3.1.1)+ L
c.3.1	Classe change (Abstract/Non-abstract)	
c.3.1.1	Non-abstract → Abstract	G+H+I+L
c.3.1.2	Abstract → Non-abstract	H+L
c.3.2	Classe deletion	
c.3.2.1	Non-abstract class	S+G+H+I
c.3.2.2	Abstract class	S+H+I
c.3.4	Class inheritance derivation	
c.3.4.1	Public → Private	S+I
c.3.4.2	Public → Protected	~H(S+I)
c.3.4.3	Protected → Private	H(S+~SG+~S I)
c.3.4.4	Protected → Public	-
c.3.4.5	Private → Public	-
c.3.4.6	Private → Protected	-
c.3.5	Class addition	-
c.3.6	Class inheritance structure	
c.3.6.1	Abstract class addition	S+G+H+I+ ie(3.1.1)+ L
c.3.6.2	Non abstract class addition	H+L
c.3.6.3	Abstract class deletion	H+ie(3.1.2)+ L
c.3.6.4	Non abstract class deletion	H+L

ANNEXE B

Modèle d'impact du changement pour Java (MICJ)

B.1 Tableau des changements structurels

Diminutif	Changement
Au niveau de la classe	
Ca	Ajout d'une classe
Cr	Retrait d'une classe
Cia	Interface : ajout
Cir	Interface : retrait
Cha	Héritage : ajout
Chr	Héritage : retrait
Ct_{na}	Type : non-abstrait à abstrait
Ct_{an}	Type : abstrait à non-abstrait
Au niveau des méthodes	
Mv_{ui}	Visibilité : public à privée
Mv_{uo}	Visibilité : public à protégée
Mv_{iu}	Visibilité : privée à public
Mv_{io}	Visibilité : privée à protégée
Mv_{ou}	Visibilité : protégée à public
Mv_{oi}	Visibilité : protégée à privée
Mt_{na}	Type : non-abstrait à abstrait
Mt_{an}	Type : abstrait à non-abstrait
Mt_{sn}	Type : statique à non-statique
Mt_{ns}	Type : non-statique à statique
Mt_{fn}	Type : final à non- final
Mt_{nf}	Type : non-final à final
Mpa	Paramètre : ajout
Mpr	Paramètre : retrait

Diminutif	Changement
Mpc_t	Paramètre : changement de type
Mpc_n	Paramètre : changement de nom
Mwa	Throws : ajout
Mwr	Throws : retrait
Mr_{vo}	Retour : void à Object
Mr_{oo}	Retour : Object à Object'
Mr_{ov}	Retour : Object à void
Ma	Méthode : ajout
Mr	Méthode : retrait
Au niveau des attributs de classes	
Av_{ui}	Visibilité : public à privée
Av_{uo}	Visibilité : public à protégée
Av_{iu}	Visibilité : privée à public
Av_{io}	Visibilité : privée à protégée
Av_{ou}	Visibilité : protégée à public
Av_{oi}	Visibilité : protégée à privée
At_{sn}	Type : statique à non-statique
At_{ns}	Type : non-statique à statique
At_{fn}	Type : final à non- final
At_{nf}	Type : non-final à final
Aa	Attribut : ajout
Ar	Attribut : retrait
Ata	Changement type attribut

B.2 Tableau des changements non-structurels

Diminutif	Changement
Nad	Ajout de déclaration
Nrd	Retrait de déclaration
Nai	Ajout d'initialisation
Nri	Retrait d'initialisation
Nas	Ajout du mot clé « super »
Naah	Ajout d'utilisation attribut hérité
Namh	Ajout appel méthode hérité
Nrah	Retrait d'utilisation attribut hérité
Nrmh	Retrait appel méthode hérité
Nrs	Retrait du mot clé « super ».
Nrm	Retrait appel méthode
Nam	Ajout appel méthode
Nrcm	Retrait corps de la méthode (remplacé par « ; »)
Nacm	Ajout corps de la méthode (remplacement de « ; »)
Nrms	Retrait appel méthode statique
Nams	Ajout appel méthode statique
Natc	Ajout d'un « try/catch »
Nrtc	Retrait d'un « try/catch »
Nar	Ajout de « return »
Nrr	Retrait de « return »
Naa	Ajout d'utilisation attribut
Nra	Retrait d'utilisation attribut
Naf	Ajout instruction d'affectation
Nrf	Retrait d'instruction d'affectation

B.3 Tableau des cibles

Cible	Description
Rmah	Méthode abstraite héritée
Rc	Constructeur
Rr	Redéfinition
Rma	Méthode abstraite
Rsm	Super méthode
Rh	Hiérarchique
Rmu	Mutateur
Rac	Accesseur
Rms	Méthode statique
Rmc	Méthode appelante
Ru	Utilise
Rah	Attribut hérité
Sr	Si redéfini
Sa	Si appel
Spu	Si public
Spi	Si privé
Spo	Si protégé
Sab	Si abstraite

B.4 Tableau des règles d'impacts

Changement	Règle d'impact
Ca	[Mpa (A)] + [Aa (A, L)] + [Nad (A, L)]
Cr	Mpr (A, H) + Ar (A, H) + Nrd (A, H) + Nri (A, H)
Cia	Ma {Rr} (L)
Cir	Mr {Rr} (L)]
Cha	Ma {Rmah} (L) + [Nas (L) {Rc}] + [Ma {Rr} (L)] + [Nas (L) {Rr}] + [Naah (L, A, H)] + [Namh (L, A, H)] + [Av_{uo} (H^S)] + [Av_{io} (H^S)]
Chr	Nrah (L) + Nrmh (L) + Nrs (L)+ [Mr {Rmah} (L)] + [Mr {Rr} (L)] + [Av_{ou} (H^S)] + [Av_{oi} (H^S)]
Ct_{na}	Nri (A, H)
Ct_{an}	Mt_{an} {Rma} (L) + [Nai (L, A, H)]
Mv_{ui}	Nrm (A, H)
Mv_{iu}	[Nam (A, H)]
Mv_{io}	[Nam (A, H)] + [Nas {Rr} (H)]
Mv_{oi}	Nrm (A, H) + Nrs {Rr} (H)
Mt_{na}	Nrcm (L) + Ma {Rr} (H) + Ct_{na} (L)
Mt_{an}	Nacm (L) + [Mr {Rr} (H)]
Mt_{sn}	Nrms (A, H) + [Nam (A, H)]
Mt_{ns}	Nrm (A,G)+[Nams (A,G)]
Mt_{fn}	[Ma {Rr} (H)]
Mt_{nf}	Mr {Rr} (H)
Mpa	Mpa || Ma (O : Sab) {Rr} (H) + Nrm && Nam (A) + [Ua] (L)
Mpr	Mpr || Ma (O : Sab) {Rr} (H) + Nrm && Nam (A) + Ur (L) || Nad (L)
Mpc_t	Mpc_t || Ma (O : Sab) {Rr} (H) + Nrm && Nam {Rh} (A, L)
Mwa	Mwa {Rsm} (H^S) + Mwa || Natc {Rmc} (A, L, H)
Mwr	[Mwr {Rsm} (H^S)] + [Mwr || Nrtc {Rmc} (A, H, L)]
Mr_{vo}	Nar (L) + Mr_{vo} {Rr} (H)+ [Nrm && Nam {Rmc} (A, H, L)]
Mr_{oo}	Mr_{oo} {Rr} (H) + Nrm && Nam {O : Rh} (A, L, H) + Nrr && Nar {Rh} (L)
Mr_{ov}	Nrr (L) + Mr_{ov} {Rr} (H) + Nrm && Nam {Rmc} (A, H, L)
Ma	Ma {Rr} (O : Sab) (H) + [Nam (L, A, H)]

B.4 Tableau des règles d'impacts (suite)

Changement	Règle d'impact
Mr	[Mr (H)] + Nrm (L, A) + [Nrm (H)]
Av_{ui}	Nra (H, A) + [Ma {Rmu} (L)] + [Ma {Rac} (L)]
Av_{uo}	Nra (A) + [Ma {Rac} (L)] + [Ma {Rmu} (L)]
Av_{iu}	[Naa (H, A)] + [Mr {Rmu}] + [Mr {Rac} (L)]
Av_{io}	[Naa (H)] + [Mr {Rmu} (L)] + [Mr {Rac} (L)]
Av_{ou}	[Naa (A)] + [Mr {Rmu} (L)] + [Mr {Rac} (L)]
Av_{oi}	Nra (H) + [Ma {Rmu} (L)] + [Ma {Rac} (L)]
At_{sn}	{Ru} (L) || Nra {Rms} (L) +Nra(A)
At_{ns}	[Mt_{ns} {Ru} (L)]
At_{fn}	[Naf (L, H, A)] + [Ma {Rmu} (L)]
At_{nf}	Nrf (L, H, A) + Mr {Rmu} (L)
Ata	Mr_{oo} {Rac} (L) + [Nra && Naa {Rh} (A, H, L)] + [Mpc_t {Rh + Rmu} (L)]
Aa	[Ma {Rac} (L)] + [Ma {Rmu} (L)] + [Naa (L, H, A)]
Ar	Mr {Rac} (L) + Mr {Rmu} (L) + Nra (L, H, A) + [Mpr {Rc} (L)]

ANNEXE C

Validation des règles d'impacts

Dans cette annexe, les changements initiaux seront surlignés en gris, les impacts prédits avec incertitude seront encadrés et les impacts prédits avec certitude seront soulignés. Les commentaires seront écrits en gris.

Changement de niveau classe

Ajout d'une classe

Règle : **Ca→**[Mpa (A)] + [Aa (A, L)] + [Nad (A, L)]

Version A

```
public class A
{
    public void methodeA()
    {
    }
}
```

Version B

```
public class A
{
    private B unB;

    public void methodeA(B deuxB)
    {
        B troisB = new B();
        System.out.println(unB);
    }
}

public class B
{
}
```

Retrait d'une classe

Règle: **Cr→**Mpr (A, H) + Ar (A, H) + Nrd (A, H)

Version A

```
public class A
{
    Private B unB;

    public void methodeA(B deuxB)
    {
        B troisB = new B();
        System.out.println(unB);
    }
}

public class B
{
}
```

Version B

```
public class A
{
    public void methodeA()
    {
    }
}
```

Ajout d'un heritage

Règle: **Cha→**Ma {Rmah} (L) + [Nas (L) {Rc}] + [Ma {Rr} (L)] + [Nas (L) {Rr}] + [Naah (L, A, H)] + [Namh (L, A, H)] + [Av_{uo} (H^s)] + [Av_{io} (H^s)]

Version A

```
public abstract class A
{
   protected String name;
   private String name2;

   public abstract void
methodA();

   public void methodAA()
   {  name = "qqchose";  }

   public void method3A(){}

   public void method4A(){}
}

public class B
{
   public B()
   {}

   public void method3A()
   {}
}
```

Version B

```
public abstract class A
{
   protected String name;
   protected String name2;

   public abstract void
methodA();

   public void methodAA()
   {  name = "qqchose";  }

   public void method3A(){}
}

public class B extends A
{
   public B()
   {  super();   }

   Public void methodA()
   {}

   Public void methodAA()
   {  name = "autre";   }

   public method3A()
   {
      super.method3A();
      name = "a";
      methode4A();
   }
}
```

Retrait d'une interface

Règle: **Cir→**[Mr {Rr} (L)]

Version A

```
public class A implements Comparable
{
   public int compareTo(Object o)
   {  return 0;   }
}
```

Version B

```
public class A
{

}
```

Retrait d'un heritage

Règle: **Chr→** Nrah (L) + Nrmh (L) + Nrs (L)+ [Mr {Rmah} (L)] + [Mr {Rr} (L)] + [Av_{ou} (H^s)] + [Av_{oi} (H^s)]

Version A

```
public abstract class A
{
   protected String name;
   protected String name2;

   public abstract void
methodA();

   public void methodAA()
   {  name = "qqchose";  }

   public void method3A(){}
}

public class B extends A
{
   public B()
   {
      super();
   }

   public void methodA(){}

   public void methodAA()
   {  name = "autre";  }

   public method3A()
   {
      super.method3A();
      name2 = "a";
      method4A();
   }
}
```

Version B

```
public abstract class A
{
   protected String name;
   private String name2;

   public abstract void
methodA();

   public void methodAA()
   {  name = "qqchose";  }

   public void method3A(){}

   public void method4A(){}
}

public class B
{
   public B()
   {}

   public void method3A()
   {}
}
```

Changement de type : non-abstrait à abstrait

Règle: $\mathbf{Ct_{na}}$→ Nri (A, H)

Version A

```
public class A
{
   public void methodA()
   {}
}

public class B
{
   public void methodB()
   {
      A unA = new A();
   }
}
```

Version B

```
public abstract class A
{
   public abstract void
methodA();
}

public class B
{
   public void methodB()
   {  A unA;      }
}
```

Changement de type: abstrait à non-abstrait

Règle: $\mathbf{Ct_{an}}$→ Mt_{an} {Rma} (L) + [Nai (L, A, H)]

Version A

```
public abstract class A
{
   Public abstract void
methodA();
}

public class B
{
   public void methodB()
   {  A unA;      }
}
```

Version B

```
public class A
{
   public void methodA() {}
}

public class B
{
   public void methodB()
   {
      A unA = new A();
   }
}
```

Ajout d'une interface

Règle: **Cia**→Ma {Rr} (L)

Version A

```
public class A
{

}
```

Version B

```
public class A implements Comparable
{
   public int compareTo(Object o)
   {  return 0;   }
}
```

Changement de niveau méthode

Changement de visibilité : publique à privée

Règle: $\mathbf{Mv_{ui}}$→Nrm (A, H)

Version A

```
public class A
{
   public void methodA()
   {}
}

public class B
{
   public void methodB()
   {
      (new A()).methodeA();
   }
}

public class C extends A
{
   public void methodC()
   {  methodeA();    }
}
```

Version B

```
Public class A
{
   private void methodA()
   {}
}

public class B
{
   public void methodB()
   {  new A();     }
}

public class C extends A
{
   public void methodC()
   {}
}
```

Changement de type : abstraite à non-abstraite

Règle: $\mathbf{Mt_{an}}$→Nacm (L) + [Mr {Rr} (H)]

Version A

```
public abstract class A
{
   public abstract void
methodA();
}

public class B Extends A
{
   public void methodA()
   {}
}
```

Version B

```
public class A
{
   public void methodA()
   { }
}

public class B Extends A
{
}
```

Changement de visibilité : privée à publique

Règle: $\mathbf{Mv_{iu}}$→[Nam (A, H)]

Version A

```
Public class A
{
   private void methodA()
   {}
}

public class B
{
   public void methodB()
   {  new A();     }
}

public class C extends A
{
   public void methodC()
   {}
}
```

Version B

```
public class A
{
   public void methodA() {}
}

public class B
{
   public void methodB()
   {
      (new A()).methodeA();
   }
}

public class C extends A
{
   public void methodC()
   {  methodeA(); }
}
```

Changement de type : statique à non-statique

Règle: $\mathbf{Mt_{sn}}$→ Nrms (A, H) + [Nam (A, H)]

Version A

```
public abstract class A
{
   public statique void
methodA()
   {}
}

public class B
{
   public void methodB()
   {
      A.methodA();
   }
}
```

Version B

```
public abstract class A
{
   public void methodA()
   {}
}

public class B
{
   public void methodB()
   {
      (new A()).methodA();
   }
}
```

Changement de visibilité : privée à protégée

Règle: $\mathbf{Mv_{io}}$→ [Nam (A, H)] + [Nas {Rr} (H)]

Version A

```
public class A
{
   private void methodA()
   {}
}

public class B
{
   public void methodB()
   {}
}

public class C extends A
{
   public void methodA()
   {}

   public void methodC()
   {}
}
```

Version B

```
public class A
{
   protected void methodA()
   {}
}

public class B
{
   public void methodB()
   {
      (new A()).methdoeA();
   }
}

public class C extends A
{
   public void methodA()
   {
      Super.methodA();
   }

   public void methodC()
   {
      methodeA();
      //Sans considérer la
      //redéfinition
   }
}
```

Changement de type : non-abstraite à abstraite

Règle: $\mathbf{Mt_{na}}$→Nrcm (L) + Ma {Rr} (H) + Ct_{na} (L)

Version A

```
public class A
{
   public void methodA()
   {}
}

public class B Extends A
{}
```

Version B

```
Public abstract class A
{
   public abstract void
methodA();
}

public class B Extends A
{
   public void methodA() {}
}
```

Changement de visibilité : protégée à privée

Règle: $\mathbf{Mv_{oi}}$→Nrm (A, H) + Nrs {Rr} (H)

Version A

```
public class A
{
   protected void methodA()
   {}
}

public class B
{
   public void methodB()
   {
      (new A()).methodA();
   }
}

public class C extends A
{
   public void methodA()
   {
      super.methodA();
   }

   public void methodC()
   {
      methodA();
      //Sans considérer la
      //redéfinition
   }
}
```

Version B

```
public class A
{
   private void methodA()
   {}
}

public class B
{
   public void methodB()
   {}
}

public class C extends A
{
   public void methodA()
   {}

   public void methodC()
   {}
}
```

Changement de type : non-final à final

Règle: $\mathbf{Mt_{nf}}$→Mr {Rr} (H)

Version A

```
public class A
{
   public void methodA()
   {}
}

public class B extends A
{
   public void methodA()
   {}
}
```

Version B

```
public class A
{
   public final void methodA()
   {}
}

public class B extends A
{}
```

Changement de type : non-statique à statique

Règle: $\mathbf{Mt_{ns}}$→ Nrm (A,G)+[Nams (A,G)]

Version A

```
public abstract class A
{
   public void methodA()
   {}
}

public class B
{
   public void methodB()
   {
      (new A()).methodA();
   }
}
```

Version B

```
public abstract class A
{
   public statique void
methodA()
   {}
}

public class B
{
   public void methodB()
   {
      A.methodA();
   }
}
```

Changement de type : finale à non-finale

Règle: $\mathbf{Mt_{fn}}$→[Ma {Rr} (H)]

Version A

```
public class A
{
   public final void methodA()
   {}
}

public class B extends A
{
}
```

Version B

```
public class A
{
   public void methodA()
   {}
}

public class B extends A
{
   public void methodA()
   {}
}
```

Ajout d'un paramètre

Règle: **Mpa→**Mpa || Ma (O : Sab) {Rr} (H) + Nrm && Nam (A) + [Ua]

Version A

```
public class A
{
   public void methodA()
   {}
}

public class B extends A
{
   public void methodA()
   {}
}

public class C
{
   public void methodC()
   {
      (new A()).methodA();
   }
}
//----Exemple 2-----------
public abstract class D
{
   public abstract void
methodD();
}

public class E extends D
{
   public void methodD()
   {}
}
```

Version B

```
public class A
{
   public void methodA(int val)
   {  val++;      }
}

public class B extends A
{
   public void methodA(int val)
   {}
   //Autre possibilité :
   //garder la méthode sans
   //paramètre et ajouter une
   //surdéfinition avec param-
   //mètre
}

public class C
{
   public void methodC()
   {
      (new A()).methodA(3);
   }
}
//----Exemple 2-------------
public abstract class D
{
   public abstract void
methodD(int val);
}

public class E extends D
{
   public void methodD(int val)
   {}
   //Autre possibilité :
   //garder la méthode sans
   //paramètre et ajouter une
   //surdéfinition avec param-
   //mètre
}
```

Retrait d'un paramètre

Règle: **Mpr→**Mpr || Ma (O : Sab) {Rr} (H) + Nrm && Nam (A) + Ur (L) || Nad (L)

Version A

```
public class A
{
   public void methodA(int
val)
   {
      val = 1;
      //Autre possibilité
      //dans la version B:
      //int val = 1
   }
}

public class B extends A
{
   public void methodA(
int val){}
   //Autre possibilité :
   //Garder la méthode avec
   //paramètre et ajouter
   //la redéfinition sans
   //paramètre
}

public class C
{
   public void methodC()
   {
      (new A()).methodA(3);
   }
}
//----Exemple 2------------
public abstract class D
{
   public abstract void
methodD(int val);
}

public class E extends D
{
   public void methodD(int
val){}
   //Autre possibilité:
   //Garder la méthode avec
   //paramètre et ajouter
   //la redéfinition sans
   //paramètre
}
```

Version B

```
public class A
{
   public void methodA()
   {}
}

public class B extends A
{
   public void methodA()
   {}
}

public class C
{
   public void methodC()
   {
      (new A()).methodA();
   }
}

//----Exemple 2------------

public abstract class D
{
   public abstract void
methodD();
}

public class E extends D
{
   public void methodD()
   {}
}
```

Changement de type d'un paramètre

Règle: **Mpc_t**→ Mpc_t || Ma (O : Sab) {Rr} (H) + Nrm && Nam {Rh} (A)

Version A

```
public class A
{
   public void
     methodA(double val){}
}

public class B extends A
{
   public void
       methodA(double val)
   {  val++;       }
}

public class C
{
   public void methodC()
   {
       (new A()).methodA(3);
       (new A())
            .methodA(3.0);
   }
}

public abstract class D
{
   public abstract void
      methodD(double val);
}

public class E extends D
{
   public void
      methodD(double val){}
}
```

Version B

```
public class A
{
   public void methodA(int
val){}
}

public class B extends A
{
   public void methodA(int
val){}
   //Autre possibilité:
   //Garder la méthode avec
   //paramètre réel et
   //ajouter la redéfinition
   //avec le paramètre de
   //type entier
}

public class C
{
   public void methodC()
   {
       (new A()).methodA(3);
       (new A()).
         methodA((int) 3.0);
   }
}

public abstract class D
{
   public abstract void
      methodD(int val);
}

public class E extends D
{
   public void methodD(int
val){}
   //Autre possibilité:
   //Garder la méthode avec
   //paramètre réel et
   //ajouter la redéfinition
   //avec le paramètre de
   //type entier
}
```

Ajout d'un lancement d'exception à la méthode

Règle: **Mwa→**Mwa {Rsm} (H^s) + Mwa || Natc {Rmc} (A, L, H)

Version A

```
public class A
{
   public void methodA()
   {}
}

public class B extends A
{
   public void methodA()
   {}
}

public class C
{
   public void methodC()
   {
      (new B()).methodA();
   }

   public void methodCC()
   {
      (new A()).methodA();
   }

   public void methodCCC()
   {
      (new B()).methodA();
   }
}
```

Version B

```
public class A
{
   public void methodA()
throws IOException
   {}
}

public class B extends A
{
   public void methodA()
throws IOException
   {}
}

public class C
{
   public void methodC()
throws IOException
   {
      (new B()).methodA();
   }

   public void methodCC()
throws IOException
   {
      (new A()).methodA();
   }

   public void methodCCC()
   {
      try{
         (new B()).methodA();
      }catch(IOException e){}
   }
}
```

Retrait d'un lancement d'exception à la méthode

Règle: **Mwr→**[Mwr {Rsm} (H^s)] + [Mwr || Nrtc {Rmc} (A, H, L)]

Version A

```
public class A
{
   public void methodA()
throws IOException {}
}

public class B extends A
{
   public void methodA()
throws IOException {}
}

public class C
{
   public void methodC()
throws IOException
   {
      (new B()).methodA();
   }

   public void methodCC()
throws IOException
   {
      (new A()).methodA();
   }

   public void methodCCC()
   {
      try{
         (new B()).methodA();
      }catch(IOException e){}
   }
}
```

Version B

```
public class A
{
   public void methodA()
   {}
}

public class B extends A
{
   public void methodA()
   {}
}

public class C
{
   public void methodC()
   {
      (new B()).methodA();
   }

   public void methodCC()
   {
      (new A()).methodA();
   }

   public void methodCCC()
   {
      (new B()).methodA();
   }
}
```

Changement de type de retour : *void* à *Object*

Règle: $\mathbf{Mr_{vo}} \rightarrow$ Nar (L) + Mr_{vo} {Rr} (H)+ [Nrm && Nam {Rmc} (A, H, L)]

Version A

```
public class A
{
    public void methodA()
    {}
}

public class B extends A
{
    public void methodA()
    {}
}

public class C
{
    public void methodC()
    {
       (new A()).methodA();
    }

    public void methodCC()
    {
       (new A()).methodA();
    }
}
```

Version B

```
public class A
{
    public int methodA()
    { return 0;   }
}

public class B extends A
{
    public int methodA()
    { return 0;   }
}

public class C
{
    public void methodC()
    {
       (new A()).methodA();
    }

    public void methodCC()
    {
       int x = (new
A()).methodA();
    }
}
```

Changement de type de retour : *Object* à *Object*

Règle: $\mathbf{Mr_{oo}} \rightarrow Mr_{oo}$ {Rr} (H) + Nrm && Nam {O : Rh} (A, L, H) + Nrr && Nar {O : Rh} (L)

Version A

```
public class A
{
    public int methodA()
    {
       return 0;
    }
}

public class B extends A
{
    public int methodA()
    {
       return 0;
    }
}

public class C
{
    public void methodC()
    {
       double x = (new
A()).methodA();
    }

    public void methodCC()
    {
       int x = (new
A()).methodA();
    }
}
```

Version B

```
public class A
{
    public float methodA()
    {
       return (float)0.0;
    }
}

public class B extends A
{
    public float methodA()
    {
       return (float)0.0;
    }
}

public class C
{
    public void methodC()
    {
       float x = (new
A()).methodA();
    }

    public void methodCC()
    {
       double x = (new
A()).methodA();
    }
}
```

Changement de type de retour : *Object* à *void*

Règle: $\mathbf{Mr_{ov}} \rightarrow$ Nrr (L) + Mr_{ov} {Rr} (H) + Nrm && Nam {Rmc} (A, H, L)

Version A

```
public class A
{
    public int methodA()
    {
       return 0;
    }
}

public class B extends A
{
    public int methodA()
    {
       return 0;
    }
}

public class C
{
    public void methodC()
    {
       (new A()).methodA();
    }

    public void methodCC()
    {
       int x = (new
A()).methodA();
    }
}
```

Version B

```
public class A
{
    public void methodA()
    {}
}

public class B extends A
{
    public void methodA()
    {}
}

public class C
{
    public void methodC()
    {
       (new A()).methodA();
    }

    public void methodCC()
    {
       (new A()).methodA();
    }
}
```

Ajout d'une méthode

Règle: **Ma→**Ma {Rr} (O : Sab) (H) + [Nam (L, A, H)]

Version A

```
public class A
{}

public class B extends A
{}

public class C
{
   public void methodC()
   {}
}
//----Exemple 2--
public abstract class D
{}

public class E extends D
{}

public class F
{
   public methodF()
   {}
}
```

Version B

```
public class A
{
   public void methodA()
   {}
}

public class B extends A
{
   public void methodA() {}
}

public class C
{
   public void methodC()
   {
      (new A()).methodA();
   }
}
//----Exemple 2--
public abstract class D
{
   public abstract void
methodD() {}
}

public class E extends D
{
   Public void methodD()
   {}
}

public class F
{
   public methodF()
   {
      (new E()).methodD();
   }
}
```

Retrait d'une méthode

Règle: **Mr→**[Mr (H)] + Nrm (L, A) + [Nrm (H)]

Version A

```
public class A
{
    public void methodA()
    {}
}

public class B extends A
{
    public void methodA(){}
}

public class C
{
    public void methodC()
    {
        (new A()).methodA();
    }
}
//----Exemple 2--
public abstract class D
{
    public abstract void
methodD() {}
}

public class E extends D
{
    public void methodD(){}
}

public class F
{
    public methodF()
    {
       (new E()).methodD();
    }
}
```

Version B

```
public class A
{}

public class B extends A
{}

public class C
{
    public void methodC()
    {}
}
//----Exemple 2--
public abstract class D
{}

public class E extends D
{}

public class F
{
    public methodF(){}
}
```

Changement de niveau attribut

Changement de visibilité : publique à privée

Règle : $\mathbf{Av_{ui}}$→Nra (H, A) + [Ma {Rmu} (L)] + [Ma {Rac} (L)]

Version A

```
public class A
{
   public int a = 0;
}

public class B extends A
{
   public void methodB()
   {  A++;  }
}

public class C
{
   public void methodC()
   {
        (new A()).a = 0;
   }
}
```

Version B

```
public class A
{
   private int a = 0;

   public int get_a()
   {  return a;   }

   public void set_a(int a)
   {  this.a = a; }
}

public class B extends A
{
   public void methodB()
   {}
}

public class C
{
   public void methodC()
   {
        (new A()).set_a(0);
   }
}
```

Changement de type : non-statique à statique

Règle: $\mathbf{At_{ns}}$→ [Mt_{ns} {Ru} (L)]

Version A

```
public class A
{
   private int a = 0;

   public void methodA()
   {
      a++;
   }
}
```

Version B

```
public class A
{
   private static int a = 0;

   public static void methodA()
   {
      a++;
   }
}
```

Changement de visibilité : publique à protégée

Règle : $\mathbf{Av_{uo}} \rightarrow$ Nra (A) + [Ma {Rac} (L)] + [Ma {Rmu} (L)]

Version A

```
public class A
{
   public int a = 0;
}

public class C
{
   public void methodC()
   {
        (new A()).a = 0;
   }
}
```

Version B

```
public class A
{
   protected int a = 0;

   public int get_a()
   {  return a;   }

   public void set_a(int a)
   {  this.a = a; }
}

public class C
{
   public void methodC()
   {
        (new A()).set_a(0);
   }
}
```

Changement de visibilité : protégé à publique

Règle : $\mathbf{Av_{ou}} \rightarrow$ [Naa (A)] + [Mr {Rmu} (L)] + [Mr {Rac} (L)]

Version A

```
public class A
{
   protected int a = 0;

   public int get_a()
   {  return a;   }

   public void set_a(int a)
   {  this.a = a; }
}

public class C
{
   public void methodC()
   {
        (new A()).set_a(0);
   }
}
```

Version B

```
public class A
{
   public int a = 0;
}

public class C
{
   public void methodC()
   {
        (new A()).a = 0;
   }
}
```

Changement de visibilité : privée à publique

Règle : $\mathbf{Av_{iu}}$→[Naa (H, A)] + [Mr {Rmu}] + [Mr {Rac} (L)]

Version A

```
public class A
{
   private int a = 0;

   public int get_a()
   {  return a;   }

   public void set_a(int a)
   {  this.a = a; }
}

public class B extends A
{
   public void methodB()
   {}
}

public class C
{
   public void methodC()
   {
        (new A()).set_a(0);
   }
}
```

Version B

```
public class A
{
   public int a = 0;
}

public class B extends A
{
   public void methodB()
   {
      a++;
   }
}

public class C
{
   public void methodC()
   {
        (new A()).a = 0;
   }
}
```

Changement de type : final à non-final

Règle: $\mathbf{At_{fn}}$→[Naf (L ,H, A)] + [Ma {Rmu} (L)]

Version A

```
public class A
{
   private final int a = 0;

   public void methodA()
   {}
}
```

Version B

```
public class A
{
   private int a = 0;

   public void methodA()
   {
      a = 3;
   }

   public void set_a(int a)
   {  this.a = a; }
}
```

Changement de visibilité : privée à protégée

Règle : $\mathbf{Av_{io}} \rightarrow$ [Naa (H)] + [Mr {Rmu} (L)] + [Mr {Rac} (L)]

Version A

```
public class A
{
   private int a = 0;

   public int get_a()
   {  return a;   }

   public void set_a(int a)
   {  this.a = a; }
}

public class B extends A
{
   public void methodB()
   {}
}

public class C
{
   public void methodC()
   {
        (new A()).set_a(0);
   }
}
```

Version B

```
public class A
{
   protected int a = 0;
}

public class B extends A
{
   public void methodB()
   {
      a++;
   }
}

public class C
{
   public void methodC()
   {}
}
```

Changement de type de l'attribut

Règle: **Ata** $\rightarrow Mr_{oo}$ {Rac} (L) + [Nra && Naa {Rh} (A, H, L)] + [Mpc_t {Rh + Rmu} (L)]

Version A

```
public class A
{
  private int a = 0;

  public void methodA()
  {   a = 3;      }

  public int get_a()
  {  return a;  }

  public void set_a(int a)
  {  this.a = a;  }
}
```

Version B

```
public class A
{
   private double a = 0;

   public void methodA()
   {  a = 3.0;    }

   public double get_a()
   {  return a;  }

   public void set_a(double a)
   {  this.a = a;  }
}
```

Changement de visibilité : protégée à privée

Règle : $\mathbf{Av_{oi}}$→Nra (H) + [Ma {Rmu} (L)] + [Ma {Rac} (L)]

Version A

```
public class A
{
   protected int a = 0;
}

public class B extends A
{
   public void methodB()
   {
      a++;
   }
}

public class C
{
   public void methodC()
   {}
}
```

Version B

```
public class A
{
   private int a = 0;

   public int get_a()
   {  return a;   }

   public void set_a(int a)
   {  this.a = a; }
}

public class B extends A
{
   public void methodB()
   {}
}

public class C
{
   public void methodC()
   {
        (new A()).set_a(0);
   }
}
```

Changement de type : statique à non-statique

Règle: $\mathbf{At_{sn}}$→ Mt_{sn} {Ru} (L) || Nra {Rms} (L) + Nra (A)

Version A

```
public class A
{
   private static int a = 0;

   public static void
methodA()
   {
      a++;
   }

   public static void
methodB()
   {
      a++;
   }
}
```

Version B

```
public class A
{
   private int a = 0;

   public void methodA()
   {  a++;  }

   public static void
methodB()
   {}
}
```

Changement de type : non-final à final

Règle: $\mathbf{At_{nf}}$→Nrf (L, H, A) + Mr {Rmu} (L)

Version A

```
public class A
{
   private int a = 0;

   public void methodA()
   {
      a = 3;
   }

   public void set_a(int a)
   {  this.a = a; }
}
```

Version B

```
public class A
{
  private final int a = 0;

  public void methodA()
  {}
}
```

Ajout d'un attribut

Règle: **Aa**→[Ma {Rac} (L)] + [Ma {Rmu} (L)] + [Naa (L, H, A)]

Version A

```
public class A
{
   public void methodA()
   {}
}
```

Version B

```
public class A
{
   private int a;

   public void methodA()
   {  a++;  }

   public void set_a(int a)
   {  this.a = a; }

   public int get_a()
   {  return a;   }
}
```

Retrait d'un attribut

Règle: **Ar→**Mr {Rac} (L) + Mr {Rmu} (L) + Nra (L, H, A) + [Mpr {Rc} (L)]

Version A

```
public class A
{
   private int a;

   public A(int a)
   {  this.a = a;        }

   public void methodA()
   {  a++;  }

   public void set a(int a)
   {  this.a = a; }

   public int get a()
   {  return a;   }
}
```

Version B

```
public class A
{
   public A(int a)
   {
      this.a = a;
   }

   public void methodA()
   {}
}
```

ANNEXE D

Code source du prototype

D.1 Spec.xml

Ce fichier contient la version XML abrégée du modèle.

```
<?xml version="1.0" encoding="UTF-8"?>

<ChangementModel>
   <ChangementType id="Cha">
      <impact type="Ma" obligation="true" predictif="true">
         <scope value = "L"/>
         <cible type="Rmah"/>
      </impact>
      <impact type="Nas" obligation="false" predictif="true">
         <scope value = "L"/>
         <cible type="Rc"/>
      </impact>
      <impact type="Ma" obligation="false" predictif="true">
         <scope value = "L"/>
         <cible type="Rr"/>
      </impact>
      <impact type="Nas" obligation="false" predictif="true">
         <scope value = "L"/>
         <cible type="Rr"/>
      </impact>
      <impact type="Naah" obligation="false" predictif="true">
         <scope value = "L"/>
         <scope value = "H"/>
         <scope value = "A"/>
      </impact>
      <impact type="Namh" obligation="false" predictif="true">
         <scope value = "L"/>
         <scope value = "H"/>
         <scope value = "A"/>
      </impact>
      <impact type="Avuo" obligation="false" predictif="true">
         <scope value = "HS"/>
      </impact>
      <impact type="Avio" obligation="false" predictif="true">
         <scope value = "HS"/>
      </impact>
   </changementType>
   <ChangementType id="Ctan">
      <impact type="Mtan" obligation="true" predictif="true">
         <scope value = "L"/>
         <cible type = "Rma"/>
      </impact>
      <impact type="Nai" obligation="false" predictif="true">
         <scope value = "A"/>
         <scope value = "H"/>
         <scope value = "L"/>
      </impact>
   </changementType>
   <ChangementType id="Mpa">
      <impact type="Mpa" or="ma" obligation="true" predictif="true">
         <scope value = "H"/>
         <cible type = "Rr"/>
```

```
         <condition type = "Sab"/>
      </impact>
      <impact      type="Nrm"      and="Nam"      obligation="true"
predictif="true">
         <scope value = "A"/>
      </impact>
      <impact type="Ua" obligation="false" predictif="true">
         <scope value = "L"/>
      </impact>
   </changementType>
   <ChangementType id="Mpr">
      <impact type="Mpr" or="ma" obligation="true" predictif="true">
         <scope value = "H"/>
         <cible type = "Rr"/>
         <condition type = "Sab"/>
      </impact>
      <impact      type="Nrm"      and="Nam"      obligation="true"
predictif="true">
         <scope value = "A"/>
      </impact>
      <impact type="Ur" or="Nad" obligation="true" predictif="true">
         <scope value = "L"/>
      </impact>
   </changementType>

   <!-- ################ Changements non-structurel ##################
-->
   <ChangementType id="Nrd">
   </changementType>
   <ChangementType id="Nad">
   </changementType>

</ChangementModel>

<scopeModel>
   <scopeType id="A"/>
   <scopeType id="H"/>
   <scopeType id="HS"/>
   <scopeType id="L"/>
</scopeModel>

<!-- ################ Cible ######################### -->
<cibleModel>
   <cibleType id="Rmah"/>
   <cibleType id="Rc"/>
   <cibleType id="Rr"/>
</cibleModel>

<!-- Le lien du modele avec les textes associés se fait par
l'intermédiaire du Id -->
<!-- Tous les id doivent être unique par catégorie -->
<text>
   <changement>
      <Ca   name = "Ca" description="Ajout d'une classe"/>
      <!--      ################      Changements      non-structurel
################## -->
      <Ua   name   =   "Ua"   description="Ajout   d'utilisation   du
paramètre"/>
```

```
      <Ur   name   =   "Ur"   description="Retrait   d'utilisation   du
paramètre"/>
   </changement>
   <scope>
      <A name="A" description="Association"/>
      <H name="H" description="Héritage"/>
      <HS name="HS" description="Héritage ascendant"/>
      <L name="L" description="Local"/>
   </scope>
   <cible>
      <Rmah name="Rmah" description="Méthode abstraite héritée"/>
      <Rc name="Rc" description="Méthode abstraite héritée"/>
      <Rr name="Rr" description="Méthode abstraite héritée"/>
   </cible>
</text>
```