Université de Montréal

OO1_correct : un environnement de restructuration des programmes objets piloté par la qualité

par

Hassan M. Chawiche

Département d'Informatique et de Recherche Opérationnelle
Faculté des arts et des sciences

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

Mars, 2002

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé :
OO1_correct : un environnement de restructuration des programmes objets piloté par la qualité

Présenté par :

Hassan M. Chawiche

a été évalué par un jury composé des personnes suivantes :

Jian-Yun Nie
.....
(Président-rapporteur)

Houari Abdelkri Sahraoui
.....
(Directeur de recherche)

Julie Vachon
.....
(Membre du jury)

Mémoire accepté le 13 mai 2002

Résumé

La restructuration joue un rôle important dans l'évolution des logiciels. Ses utilisations s'étendent de l'apparemment insignifiant, tel que le changement du nom d'un attribut d'une classe, au profond, tel que le réajustement ultérieur de patron de conception dans un système existant. Malgré son importance, le manque d'outils qui aident à la détection automatique des situations où une transformation particulière pourra être appliquée force beaucoup des programmeurs de procéder à une restructuration manuelle, une tâche pour le moins ardue et sujette aux erreurs.

Dans ce travail, nous avons développé l'outil OO1_correct, un environnement de détection et de correction automatique des défauts de conception des systèmes à objets. Le comportement de cet outil est similaire à celui d'un correcteur de style d'un programme de traitement de texte. En effet, dès qu'une situation symptomatique est localisée dans une classe, par l'entremise d'un modèle flou de prédiction de la qualité, l'outil suggère des transformations susceptibles d'améliorer la qualité, en laissant à l'utilisateur le choix d'accepter ou de refuser cette suggestion. Le principe d'une prescription se base sur l'établissement d'une relation entre les transformations et l'amélioration de la qualité. La dérivation d'une telle relation se base sur l'hypothèse stipulant que si une bonne conception correspond à une bonne combinaison des valeurs des métriques, alors il y a de fortes chances qu'une bonne combinaison de valeurs des métriques correspond à une bonne conception du système [SAH00].

L'outil a été évalué sur un système de taille moyenne, écrit entièrement en Java, et les résultats obtenus sont très encourageants.

Mots clés : restructuration, refactoring, modèle prédictif, modèle de qualité, règle de prédiction, logique floue, détection automatique de défauts, design.

Abstract

Refactoring is an important part of the evolution of reusable software and frameworks. Its uses range from the seemingly trivial, such as renaming program elements, to the profound, such as retrofitting design patterns into an existing system. Despite its importance, the lack of tool that helps in detection of design flaws where a particular transformation can be applied, forces programmers to refactor programs by hand. This can be tedious and error prone.

In this thesis, we present the OO1_correct tool that creates an environment for automatic detection and correction of design flaws in objects oriented systems. The behaviour of this tool is analogous to a linguistic assistant for a text processor. Indeed, once a symptomatic situation is detected in a class using a fuzzy quality model, the tool suggests some transformations that improve the quality, while preserving its behaviour, and letting the ultimate choice to the user either to accept or refuse these transformations. The principle of the prescription is based on the establishment of relationship between transformations and the improvement of the quality. The derivation of such relation is based on the hypothesis that if a good design corresponds to a good combination of the metrics values, then there are strong probabilities that a good combination of metrics values corresponds to a good design [SAH00].

Our approach has been evaluated on a system written entirely in Java, and the obtained results are very encouraging.

Keywords: refactoring, predictive model, quality model, fuzzy rule, fuzzy logic, automatic detection of design flaws, object oriented system, design.

Tableau de matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

CIS	Class Interface Size
DAC	Data Abstraction Coupling (1)
DAC'	Data Abstraction Coupling (2)
DAM	Data Access Metric
DIT	Depth In Tree
GÉLO	Laboratoire de GÉnie LOgiciel
HTML	Hyper Text Market Language
IHI	Inheritance Hierarchy Inference
IM	Information Model
JFC	Java Foundation Classes
JVM	Java Virtual Machine
LCOM	Lack of Cohesion in Methods
NAA	Number of Available Attribute
NAI	Number of attributes inherited
NAM	Number of Available Method
NMA	Number of Method Added
NMI	Number of Method Inherited
NMO	Number of Method Overridden
NOC	Number Of Children
NOP	Number of Polymorphic Method
NPA	Number of Public Attribute
NPM	Average of the number of parameters per method

NTA	Number of Total Attribute
NumExp	Number of conditional Expressions converted to methods
NumFCL	Number of Factorized Classes
NumMA	Number of Method Abstracted
NumMATR	Number of Moved Attributes to component-abstract class
NumMC	Number of Method Created
NumPA	Number of Parameter Added
NumSCL	Number of new Subclasses
OAM	Operation Access Metric
OCAEC	Others Class-Attribute Exported Coupling
OCAIC	Other class-attribute imported coupling
OCMEC	Other Class-Method Exported Coupling
OCMIC	Other Class-Method Imported Coupling
OO	Object Oriented (Orienté Objet)
Stby	Stability
UML	Unified Modeling Language (langage unifié de modélisation)
XML	Extensible Markup Language

Dédicace

À mes très chers parents Mohamad et Raghda

À la mémoire de mon meilleur ami Ahmad

Remerciements

Mes grands remerciements reviennent à Monsieur Houari A. Sahraoui, professeur à l'Université de Montréal, pour m'avoir donné l'occasion de faire partie du groupe GÉLO, ainsi que pour avoir supervisé ce travail. Sa grande disponibilité, sa persévérance, son souci du détail ainsi que son encouragement m'ont beaucoup aidé tout le long de ce travail. Que ce mémoire soit le modeste témoignage de ma reconnaissance et de mon admiration.

Je remercie également mes amis du groupe GÉLO de l'Université de Montréal.

Je remercie tous les membres de ma famille et spécialement mes parents Mohamad et Raghda qui m'ont soutenu tout au long de mes études et qui ont fait en sorte, par leur amour, leur tendresse, que je puisse avoir les meilleures conditions possibles pour que je termine mes études supérieures. Qu'ils trouvent ici toute ma gratitude et mon amour pour eux ! Mes pensées vont aussi à mon grand ami Ahmad décédé en janvier dernier.

Finalement, je suis reconnaissant à Dieu pour m'avoir accordé les talents et les occasions qui ont rendu ce travail possible.

« Que soit issue de vous une communauté qui appelle au bien, ordonne le convenable et interdit le blâmable; car ce seront eux qui réussiront. » (Le Coran, 3:l04)

Chapitre 1

INTRODUCTION

1.1 MOTIVATIONS

Durant le cycle de vie d'un système orienté objet, les besoins d'origine évoluent, les besoins des utilisateurs changent et l'environnement matériel et logiciel du système subit des mises à jour. De ce fait, des modifications, parfois importantes, de l'application sont faites. Au fil du temps, cette maintenance corrective, évolutive ou adaptative a pu modifier la spécification et la structure des programmes ou, encore plus grave, a pu introduire des anomalies dans la conception : les systèmes deviennent difficilement compréhensibles. Leur maintenance devient de plus en plus coûteuse et complexe. Il est donc nécessaire d'envisager une restructuration de ces systèmes afin d'en améliorer la compréhension et l'évolution.

Or, la détection manuelle des situations symptomatiques de conception où une ou plusieurs transformations pourront être utilisées comme prescription, est souvent compliquée, dès que la taille du système devient importante. Un outil qui automatise les tâches de détection et de correction des défauts de conception est une façon d'adresser la complexité et de réduire le risque d'inadvertance dans le processus d'amélioration de la qualité. Ceci permet ainsi au programmeur de se concentrer sur des questions de conception plus importantes.

Deux types de travaux peuvent contribuer à l'automatisation des processus d'amélioration de la qualité : l'estimation de la qualité et les restructurations

automatiques du logiciel. Dans la plupart des cas, ces deux approches ont été traitées indépendamment l'une de l'autre. En effet, plusieurs travaux sur l'estimation de la qualité proposent des solutions permettant de détecter des anomalies de conception à l'aide de modèles prédictifs (voir [BAS96], [SAH97], [MAO98] et [ELE01]). Cependant, ils ne suggèrent pas ou ne permettent pas la suggestion d'alternatives pour la correction de ces anomalies. De plus, les modèles prédictifs utilisés sont écrits en logique classique où une proposition ne peut être que 'vraie' ou 'fausse'. L'utilisation de cette logique ne tolère pas les imprécisions dans le processus d'estimation et ne tient pas compte de la notion imprécise de la qualité, qui est bien basée sur des connaissances subjectives (informations approximatives et parfois incertaines).

Par ailleurs, divers travaux proposent des restructurations élémentaires ou complexes permettant d'améliorer la qualité d'un système orienté objet, tout en préservant son comportement (voir [CAS91] et [OPD92]). Toutefois, il est difficile de détecter les endroits où de telles restructurations peuvent être appliquées et quel est leur impact sur la qualité.

Dans ce travail, nous avons développé une technique qui combine ces deux familles de travaux. En effet, nous proposons une approche pour détecter automatiquement des situations où une restructuration particulière pourra être appliquée afin d'améliorer la qualité d'un système. Le processus de détection est basé sur des relations de cause à effet établies entre des transformations de haut niveau et quelques métriques logicielles[1], en utilisant des modèles prédictifs flous. Ces types de modèles sont plus souples, et expriment davantage les perceptions et le savoir-faire des experts en matière de prédiction de la qualité. La restructuration est alors pilotée par la sélection de variations de valeurs de quelques métriques pour éviter des situations d'anomalies.

La mise en œuvre de notre approche est réalisée par l'entremise de l'outil OO1_Correct, développé dans le cadre de ce travail. Ce prototype est semi-automatique, car il nécessitera l'intervention humaine et la connaissance du domaine du système à traiter. Cet

[1] Voir Chapitre 3, section Définition des Métriques.

outil nous a permis de prouver l'efficacité de notre approche, et les résultats obtenus sont particulièrement encourageants.

1.2 CONTRIBUTIONS PRINCIPALES

Ce travail apporte deux contributions majeures. Premièrement, l'approche de restructuration présentée dans ce mémoire s'appuie sur un concept peu exploité à des fins d'amélioration de la qualité jusqu'à présent. Un diagnostic automatique des anomalies de conception, pour lesquelles des transformations seront suggérées pour les éliminer, constitue une approche innovatrice qui n'a guère été explorée dans le domaine du génie logiciel. Loin d'être parfaite, notre recherche promet néanmoins d'ouvrir de nouvelles possibilités en termes d'aide à l'automatisation de restructuration de logiciels orientés objet, et dans laquelle d'autres alternatives de conception, comme le patron de conception (ou "*design pattern*") pourront s'intégrer facilement.

La deuxième contribution majeure de ce travail consiste en l'utilisation de la logique floue dans les modèles de prédiction de la qualité. En effet, la logique floue est une approche qualitative de la réalité qui s'inspire du savoir-faire technique dont dispose un expert. Un tel modèle permet d'intégrer dans le processus d'évaluation, une sorte d'appréciation qualitative plutôt que quantitative de la qualité, permettant ainsi la reproduction de connaissance ou le jugement de l'expert du domaine. En outre, la définition des fonctions d'appartenance sur les métriques a pour effet d'augmenter les possibilités de changements de valeurs de ces métriques (voir Chapitre 4, section changements offerts par les formes de fonctions d'appartenance). Ceci permet à un grand nombre de restructurations d'être appliquées pour améliorer la qualité.

1.3 STRUCTURE DE CE MÉMOIRE

Le chapitre 2 présente plusieurs travaux effectués dans les domaines d'estimation de la qualité et d'automatisation de la restructuration. La mesure des impacts des quelques

restructurations de bas et de haut niveau sur les métriques est présentée dans le chapitre 3. L'approche sur laquelle se base notre outil est détaillée dans le chapitre 4. La mise en œuvre de notre technique est abordée dans le chapitre 5. Le chapitre 6 porte sur une étude expérimentale. Finalement, dans le chapitre 7 nous dressons le bilan des travaux effectués et quelques perspectives de recherche.

Chapitre 2

TRAVAUX EFFECTUÉS DANS LE DOMAINE DE RESTRUCTURATION

L'étude de la qualité, qu'elle soit pour des systèmes procéduraux ou orientés objet, a donné naissance à une série de travaux. Ces travaux peuvent être divisés en deux grandes catégories. La première étudie la corrélation entre des caractéristiques de qualité et des attributs quantitatifs d'un logiciel, comme par exemple la maintenabilité avec des métriques structurelles. La deuxième catégorie se situe au niveau de la restructuration de logiciel et de la suggestion d'alternatives de conceptions pour améliorer la qualité du système.

Deux sections majeures constituent le chapitre actuel. La première section (2.1) s'intéresse aux travaux de validation de métriques comme moyen fiable de prédiction de la qualité. Quant à la deuxième partie (2.2), elle se consacre aux travaux qui aident à l'automatisation de la restructuration logicielle.

2.1 TRAVAUX SUR L'ESTIMATION DE LA QUALITÉ

Si les composantes logicielles défaillantes peuvent être détectées au début du projet de développement, une correction, telle qu'une reprise de conception, peut être effectuée [ELE01]. Dans les applications orientées objet, les modèles de prédiction, utilisant des métriques de conception, peuvent servir à détecter les classes défectueuses dès le début.

Plusieurs travaux se sont intéressés à l'estimation de la qualité à l'aide de métriques. Dans ce qui suit, nous allons présenter les travaux que nous jugeons les plus pertinents.

2.2.1 Basili et al.

Basili et al. [BAS96] ont effectué une étude empirique sur les métriques orientées objet proposées par Chidamber et Kemerer [CHI94] (voir Tableau 1 ci-dessous). Cette étude, menée par 8 équipes d'étudiants de l'université du Maryland, visait à vérifier que les métriques proposées prédisaient la propension d'une classe à avoir des erreurs lors de la phase de conception des systèmes orientés objet. L'étude a conclu que 5 des 6 métriques proposées semblent être des prédicateurs utiles (la métrique LCOM étant invalidée). Les auteurs préconisent une validation plus représentative de ces métriques dans le cadre de véritables projets industriels.

Symbole	Nom de la métrique
WMC	Weighted Methods Per Class (Méthodes pondérées par classe)
DIT	Depth of Inheritance Tree (Profondeur de l'arbre d'héritage)
NOC	Number Of Children (Nombre d'enfants)
CBO	Coupling Between Objects (Couplage entre les classes objet)
RFC	Response For a Class (Réponse d'une classe)
LCOM	Lack of Cohesion in Methods (Manque de cohésion des méthodes)

Tableau 1. Les métriques définies par Chidamber et Kemerer

2.2.2 Briand et al.

Briand et al. [BRI97] ont défini un ensemble de métriques de couplage de systèmes orientées objet, au niveau de la conception. Dans cette étude, ils tiennent compte des différents mécanismes propres au langage C++ (classe amie, spécialisation et agrégation). Trois modalités sont considérées :

i) le type de couplage : classe-attribut, classe-méthode et méthode-méthode,

ii) la relation entre classes couplées : hiérarchique, classe amie ou autres,

iii) la localisation de l'impact : couplage importé d'une classe ancêtre ou dont ladite classe est amie ou exportée vers une classe descendante ou amie.

Cela mena les auteurs à proposer 24 métriques de couplage (Tableau 2 ci-dessous en montre une partie) qui furent validées empiriquement, en analysant la relation de cause à effet avec la probabilité de détection d'erreurs dans les classes. Le processus de validation montra que certaines de ces 24 métriques pouvaient être des indicateurs de la qualité de la conception d'un système orienté objet. Outre la dérivation purement statique des métriques, la taille du système sur lequel ces métriques ont été testées est trop petite (moins que 100 classes) et peu représentative. C'est pourquoi une réplication de cette étude sur d'autres systèmes industriels ou de grande taille semble fortement recommandée pour une éventuelle généralisation des résultats obtenus.

Symbole	Nom de la métrique
ACAIC	Ancestors Class-Attribut Import Coupling (Couplage classe- attribut importé des classes ancêtres)
IFCAIC	Inverse Friend Class-Attribut Import Coupling (Couplage inverse classe- attribut importé des classes amies)
OCMEC	Others Class-Method Export Coupling (Couplage classe-méthode exporté vers les autres classes)
OMMEC	Others Method -Method Export Coupling (Couplage méthode-méthode exporté des autres classes)
DCMEC	Descendant Class-Method Export Coupling (Couplage classe-méthode exporté vers les descendants)

Tableau 2. Liste partielle de métriques définies par Briand et al.

2.2.3 El-Emam et al.

El-Emam et al. [ELE01] ont fait une étude dans laquelle des métriques de conception orientées objet ont été utilisées pour construire des modèles prédictifs de la qualité. Les auteurs ont utilisé des données recueillies d'une application Java commerciale pour construire un modèle de prédiction. Ce modèle a ensuite été validé sur une version ultérieure de la même application. Les résultats indiquent que le modèle de prédiction est d'une grande précision. De plus, il est observé qu'une métrique de couplage d'exportations a l'association la plus prononcée avec la prédisposition aux défaillances. Ceci révèle une caractéristique structurelle, qui peut être symptomatique, d'une classe ayant une grande probabilité de défaillances latentes.

2.2.4 Demeyer et al.

Sur les bases d'une étude de l'environnement VisualWorks, un cadre de conception orienté objet (ou "*framework*") pour la construction d'interfaces graphiques dans l'environnement de Smalltalk, Demeyer et al. [DEM99] ont conclu que les métriques basées sur la taille des entités et de l'héritage ne sont pas fiables pour détecter des défauts de conception. Pourtant, ces mêmes métriques sont utiles pour mesurer les différences entre deux versions successives de même cadre d'application, ainsi que pour mesurer sa stabilisation.

2.2 TRAVAUX SUR L'AUTOMATISATION DE LA RESTRUCTURATION

L'une des raisons pour lesquelles les programmes ne sont pas restructurés manuellement est que n'importe quel changement au système court le risque d'introduire des défauts au système. Un outil qui automatise la mise en œuvre des restructurations devrait garantir que ses exécutions préservent le comportement d'un programme. Ainsi, une motivation importante pour automatiser les restructurations est d'assurer que des défauts ne sont pas introduits dans une application.

Dans cette partie, nous allons aborder les travaux effectués dans le domaine de la restructuration de logiciels, et plus particulièrement ceux qui servent à les automatiser.

2.2.1 Casais

En s'inspirant de travaux effectués dans le domaine des bases de données orientées objet, Eduardo Casais [CAS91] ébauche une série de modifications élémentaires ou primitives applicables à un objet. Toute mise à jour du schéma de la base de données peut être divisée en une série de mises à jour primitives. Chaque mise à jour primitive peut alors être analysée, afin de vérifier qu'elle préserve des invariants de classes prédéfinis.
Les invariants de classes peuvent être, par exemple, des invariants de représentation exigeant que les propriétés d'un objet reflètent celles de sa classe, ou bien des invariants de noms distincts exigeant que les classes, méthodes et variables possèdent des noms différents.

Casais souligne l'importance d'établir une série complète de primitives afin que toute transformation puisse être décomposée à l'aide de cette série. La pertinence d'une telle série va dépendre des invariants à préserver et sera décidée après une étude méticuleuse de l'effet produit par chacune des primitives.

2.2.2 Opdyke

L'un des travaux les plus intéressants et qui aident à l'automatisation de restructurations est celui d'Opdyke [OPD92], [OPD93]. Inspiré des travaux de Banerjee et al. [BAN87], concernant l'évolution du schéma de base de données orientées objet, Opdyke a élaboré la notion de transformation (restructuration ou "*refactoring*"), supportant l'évolution et la réutilisation de cadres d'application, sans changer toutefois leur comportement.
La préservation du comportement du programme est assurée par l'intermédiaire de sept invariants, définis comme suit :

i) Unicité de super-classe : après une restructuration, une classe doit toujours avoir, au plus, une seule super-classe directe et sa super-classe ne doit pas être parmi ses sous-classes.

ii) Noms distincts de classes : après une restructuration, chaque classe doit avoir un nom unique et elle ne peut pas être emboîtée. En d'autres termes, les classes sont visibles partout dans la hiérarchie.

iii) Noms distincts des membres d'une classe : après une restructuration, tous les attributs et toutes les méthodes d'une classe doivent avoir des noms distincts.

iv) Les variables membres héritées ne pourront pas être redéfinies : une variable héritée d'une super-classe ne peut être redéfinie ultérieurement dans l'une de ses sous-classes.

v) Les signatures de fonctions membres redéfinies[2] sont compatibles : après une restructuration, si une fonction définie dans une super-classe est redéfinie dans une de ses sous-classes, alors tous les attributs de ces deux fonctions, sauf le corps de fonction, doivent être compatibles.

vi) L'affectation est "*Type-Safe*[3]": après une restructuration, le type d'une expression affectée à une variable doit être une instance du type de variable en question.

vii) L'équivalent sémantique d'opérations et des références : l'équivalent sémantique est défini comme suit : si la fonction *main* [4]*()* est appelée deux fois, l'une avant et l'autre après la restructuration, avec les mêmes ensembles d'entrées, alors les deux ensembles de valeurs de sortie devront être identiques.

Un ensemble de 26 restructurations de bas niveau (voir Chapitre 3, Section 3.2.1) a été mis en place. Pour chacune d'elles, une ou plusieurs préconditions ont été définies, afin de garantir la préservation du comportement du programme. La plupart de ces

[2] Dans le langage de programmation C++, deux fonctions ont des signatures compatibles si les types de leurs arguments et de retour sont identiques.
[3] La terminologie originale anglaise est utilisée dans le texte pour éviter l'introduction d'imprécisions syntaxiques et sémantiques.
[4] *main ()* désigne la méthode de contrôle d'accès au programme.

restructurations sont très simples, voire triviales, et consiste à vérifier que leur application n'altère pas le comportement du programme. Par exemple, pour la restructuration de bas niveau « suppression d'un attribut d'une classe », une précondition de cette transformation est que l'attribut en question n'est pas référencé.

L'ensemble de transformations élémentaires est utilisé pour décomposer des restructurations plus complexes, sans introduire d'erreurs dans le système ni modifier le comportement du programme. La préservation du comportement est garantie pour chaque restructuration de haut niveau puisque les restructurations de bas niveau qui la composent préservent le comportement.

Un exemple d'une la restructuration de haut est la « création d'une classe abstraite » (cf. figure 1 ci-dessous). Cette transformation peut être composée des restructurations élémentaires suivantes:

- *création d'une classe,*
- *changement de super-classe,*
- *ajout d'un paramètre à une méthode,*
- *création d'une méthode,*
- *remplacement d'un segment de code par un appel de méthode.*

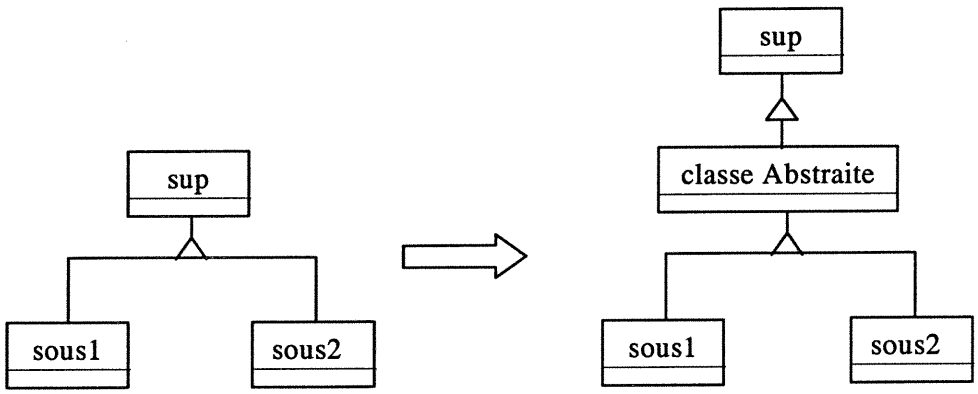


Figure 1. La création d'une classe abstraite

Les travaux d'Opdyke ne permettent pas de spécifier l'endroit du système là où une restructuration donnée peut prendre, ni de mesurer l'impact d'une telle restructuration sur les autres composants de l'application.

2.2.3 Roberts

S'inspirant des travaux d'Opdyke [OPD93], Roberts illustre dans sa thèse [ROB99] les façons de rendre l'automatisation de restructuration plus pratique, fiable et efficace. Pour ce faire, il a étendu la définition d'une restructuration, en lui ajoutant une post-condition (ou assertion). Pour que le résultat d'une restructuration soit validé, les programmes doivent satisfaire cette post-condition. L'assertion est utilisée, entre autres, pour réduire les tâches de vérification et d'analyse des préconditions d'une restructuration, pour dériver les préconditions de restructurations composites et pour établir les dépendances entre les restructurations.

Les préconditions d'une restructuration composite sont dérivées des préconditions des modifications atomiques qui composent cette restructuration. Quant aux dépendances entre les restructurations, elles sont utilisées pour supporter l'annulation d'une restructuration en cours (*undo*), le parallélisme des restructurations (*parallelizing refactoring*) et le fusionnement des restructurations dans un environnement de programmation multiple.

Les travaux de Roberts ont donné naissance à l'outil d'automatisation des transformations dans l'environnement Smalltalk, "*The Refactoring Browser*" [ROB97], dont l'interface principale est illustrée dans la figure 2 ci-dessous.

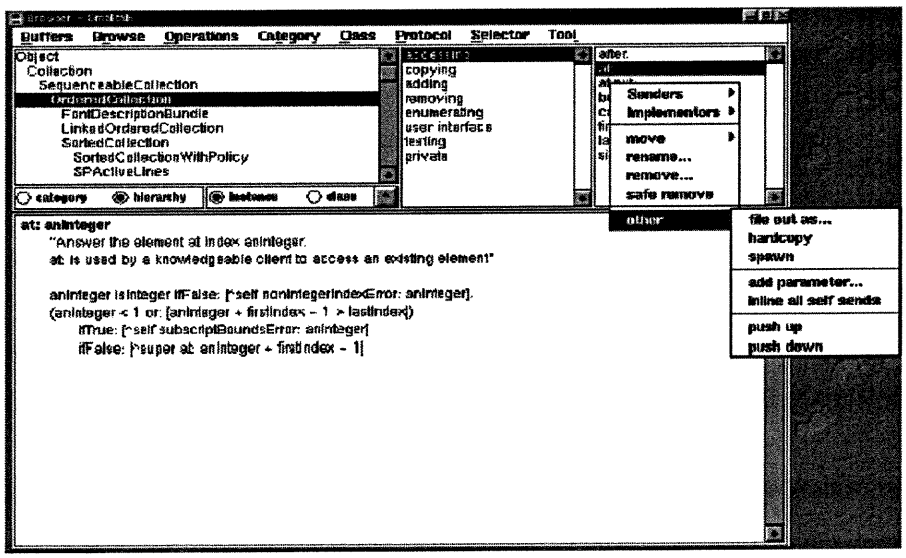


Figure 2. L'interface principale de "Refactoring Browser"

2.2.4 Moore

Moore [MOO96] présente une approche algorithmique de la restructuration et de l'évolution de la hiérarchie de classes. L'algorithme IHI ("*Inheritance Hierarchy Inference*") qu'il propose permet d'inférer la hiérarchie d'héritage à partir des caractéristiques (ou "*features*") communes entre les objets sur lesquels l'algorithme est appliqué, tout en préservant leurs comportements externes. La mise en application de cette approche s'est fait au moyen de l'outil *Guru*.

Guru prend une collection d'objets[5], qui n'appartiennent pas nécessairement à un même diagramme de classes, ni reliés par des relations d'héritage et les restructure dans une nouvelle hiérarchie (voir figure 3 ci-dessous), dans laquelle les cinq critères suivants seront vérifiés :

i) Optimisation du partage de caractéristiques communes : chaque caractéristique doit apparaître une seule fois dans le diagramme de classes. Ceci permet d'obtenir une hiérarchie compacte et facile à maintenir ;

ii) Minimisation de classes incorporant une abstraction générale ;

iii) Cohérence des relations d'héritage : si un ensemble d'objets hérite des deux classes *C* et *D*, alors *C* doit hériter (directement ou indirectement) de la classe *D* ;

iv) Les relations d'héritage transitives sont implicites ;

v) Les objets initiaux correspondent à des feuilles dans la hiérarchie d'héritage inférée.

En comparaison avec d'autres approches, l'algorithme maximise le partage des méthodes entre classes au lieu de factoriser les codes ou les méthodes communs entre classes dans une super-classe abstraite.

Les travaux de Moore ne s'approchent pas directement des problèmes liés à la maintenance ou à la compréhension de programme. Néanmoins, l'effet de réduire la

[5] L'algorithme a été testé sur des objets écrits en langage de programmation Self [UNG87].

complexité de programme et d'augmenter la cohérence entre ses parties aura un impact positif sur la compréhension totale du système.

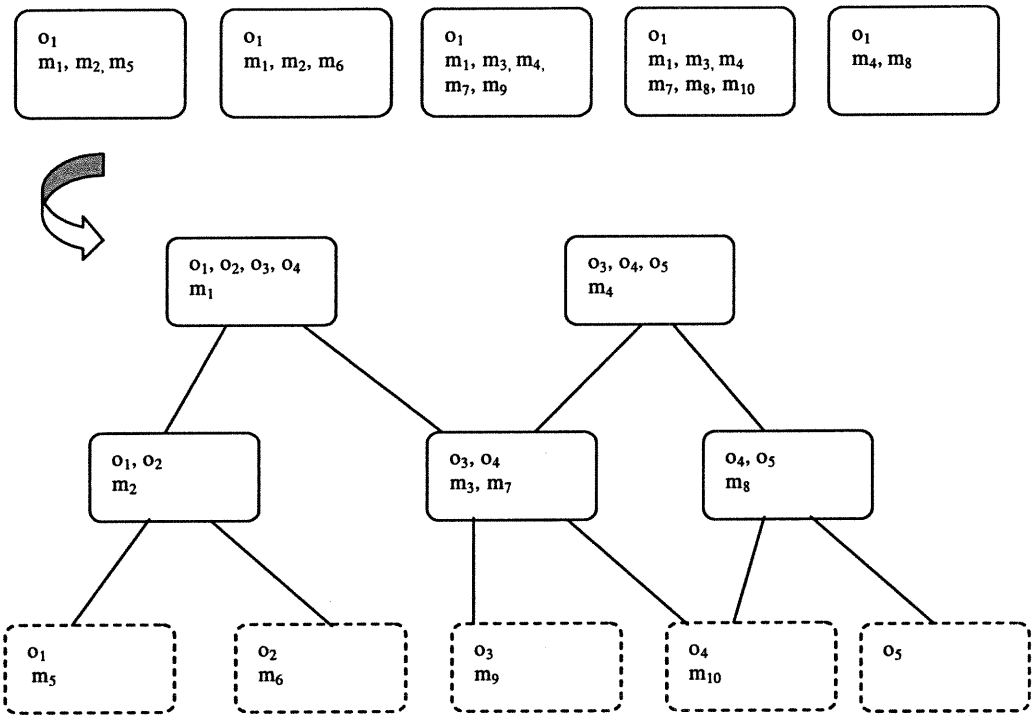


Figure 3. Un ensemble d'objets restructurés au moyen de l'outil *Guru*

2.2.5 Miceli et al.

Le travail présenté par Miceli et al. dans [MIC99b] consiste en l'étude de l'utilisation de métriques statistiques, dans l'objectif de suggérer des transformations susceptibles d'améliorer la qualité des systèmes orientés objet. La technique proposée permet la détection et la correction des anomalies au moyen de modèles de prédiction classiques et de restructurations.

Selon les modèles de qualité utilisés, une classe peut avoir seulement une classification (ou qualité) binaire, une classification 0 correspondant à une bonne qualité et une classification 1 à une mauvaise qualité. Améliorer une classe revient à changer sa classification, ce qui signifie que le résultat de l'évaluation d'une ou plusieurs règles

devra changer ou encore que la valeur d'une ou plusieurs métriques se verra modifiée adéquatement.

La variation de valeur d'une métrique est donnée sous la forme d'une règle du premier ordre. Les prémisses d'une règle précisent le contexte dans lequel la variation est valide, alors que la conclusion donne la variation de valeur de métrique en question.
Cette technique de prédiction, appliquée à un ensemble de classes, ne permet pas de les classifier selon leur degré d'appartenance aux ensembles « bonne qualité » et « mauvaise qualité ». En effet, selon la règle suivante[6] : NMI[7] > 22 ⇒ class 1, toutes les classes dont la valeur de NMI dépasse le chiffre 22 (23, 24, etc.) ont une mauvaise qualité[8]. En d'autres termes, si deux classes dont les valeurs de NMI sont respectivement égales à 23 et 43, alors ces deux classes ont une même qualité et nécessitent le même effort de maintenance, ceci sans toutefois nuancer l'écart qui existe entre 22 et 23 et entre 22 et 43. Ce problème mériterait d'être considéré d'une façon systématique.

2.2.6 O'Cinnéide et al.

O'Cinnéide et al. [OCI99] présentent une méthodologie pour l'automatisation de l'application des patrons de conception dans la ré-ingénierie. Ce travail s'inspire de celui d'Opdyke qui s'intéresse au "refactoring" des programmes en C++ [OPD92]. En effet, les deux définissent de la même manière les pré- et post-conditions des transformations pour démontrer leur préservation du comportement des programmes. Les transformations sont décomposées en mini-transformations, qui sont exprimées en termes d'un "refactoring" à un niveau plus bas. D'après leur article, il apparaît jusqu'à maintenant qu'ils n'ont travaillé qu'avec un seul patron (le patron "Factory" [GAM95]).

---

[6] Tirée du modèle de prédiction présenté dans [MIC99b].
[7] "Number of Methods Inherited" ou le nombre de méthodes héritées.
[8] Même remarque dans le cas d'une bonne qualité.

2.2.7 Tokuda et al.

Tokuda et al. [TOK99] montrent que plusieurs applications orientées objet peuvent être restructurées automatiquement en utilisant des patrons de conception comme alternatives de design. Dans ce travail, un patron de conception est vu comme étant un ensemble de modifications élémentaires. L'emphase a été mise sur la définition d'une restructuration. Les auteurs affirment que les sept invariantes définit par Opdyke, dans [OPD92], ne sont pas suffisantes pour garantir que la mise en application d'une restructuration préserve le comportement des programmes écrit en C++. C'est pourquoi, les invariants supplémentaires suivants ont été proposés :

i) Implantation de fonctions virtuelles : une classe concrète ne peut pas définir une fonction virtuelle.

ii) Préservation des objets agrégats : si un programme dépend des propriétés des objets agrégats, alors ces objets doivent être conservés.

iii) L'instanciation n'a pas des effets secondaires : si une restructuration peut changer l'ordre dans laquelle les classes sont instanciées, alors le constructeur ne peut avoir aucun effets secondaires au-delà d'initialiser l'objet crée.

Dans ces travaux, l'automatisation de restructuration concerne la vérification d'invariantes des restructurations et les modifications à apporter au code source du programme. En effet, lors de la restructuration d'une application, l'utilisateur doit s'occuper de la sélection de patrons à appliquer et l'identification de composantes de l'application où ces patrons peuvent être implantés. Il est clair que cette approche ne permet pas de détecter automatiquement des situations où un patron prend place, ni la sélection automatique de patrons à appliquer.

2.3 CONCLUSION

Dans ce chapitre, nous avons présenté les deux types de travaux qui peuvent contribuer à l'amélioration de la qualité des systèmes à objets : la prédiction de la qualité et l'utilisation de restructurations pour suggérer des alternatives de conception.

Les travaux d'estimation de la qualité, présentés dans les sections précédentes, sont distincts de notre approche dans la mesure où ils s'intéressent simplement à la détection ou localisation des classes défectueuses, sans toutefois suggérer des alternatives pour les corriger. En outre, les modèles de qualité utilisés ne tolèrent pas les imprécisions dans le processus d'estimation ; ils ne prennent pas en compte la notion imprécise de la qualité, qui est basée sur des connaissances subjectives dans la plupart des cas.

D'autre part, les travaux sur l'automatisation des restructurations ne permettent pas de spécifier les situations où une transformation peut prendre place, ni de mesurer son impact sur les autres composantes du système.

Dans le cadre de notre travail, nous utilisons des restructurations pour corriger les anomalies détectées à l'aide des modèles de prédiction flous. L'avantage d'utiliser des règles floues réside dans le fait que ces dernières sont semblables à celles utilisées dans le raisonnement humain, qui est par nature incertain. De plus, la technique de prédiction s'intéresse à la direction de changement de la qualité d'une classe, afin d'éviter les cas où l'on diminuerait la qualité au lieu de l'accroître.

Chapitre 3

MESURE DE L'IMPACT DES RESTRUCTURATIONS SUR LES MÉTRIQUES

D'une façon générale, la gestion adéquate de n'importe quel processus exige la quantification, la mesure et la représentation en un modèle. Le terme « métrique » est devenu une expression courante dans la littérature en génie logiciel. En effet, les métriques logicielles fournissent une base quantitative pour le développement et la validation des modèles de développement de logiciels. Elles peuvent être employées, entre autres, pour mesurer et améliorer la productivité et la qualité logicielle [JAC97].

Ce chapitre est organisé comme suit : tout d'abord, nous présenterons les métriques orientées objet utilisées dans ce mémoire. Ensuite, nous donnerons la définition d'une restructuration, en passant par ses deux variantes, celles de bas et de haut niveau. Par la suite, nous étudierons l'influence de ces restructurations sur les métriques, ce qui permettra d'établir des relations de cause à effet entre les restructurations en question et les valeurs des métriques concernées. Ces relations ont une grande utilité, puisqu'elles guident les suggestions de transformations lors de la détection des anomalies de conception dans une classe.

3.1 LES MÉTRIQUES UTILISÉES DANS CE MÉMOIRE

De très nombreuses propositions de métriques pour les systèmes orientés objet ont vu le jour durant les dernières années (voir [LIW93], [CHI94] et [BRI96]). L'utilisation de ces métriques réduit la subjectivité, dans l'évaluation de la qualité logicielle, en fournissant une base quantitative pour prendre des décisions. En effet, les métriques permettent de réduire l'écart existant, d'un point de vue technique, entre les informaticiens et les décideurs. Elles n'ont pas la vocation de remplacer les méthodes, mais plutôt celle de les renforcer. Nous devons donc mesurer la qualité des produits logiciels si nous voulons l'améliorer.

Bien que l'on trouve dans la littérature d'autres métriques qui permettent de mesurer la stabilité[9] des systèmes, nous nous sommes restreints aux métriques qui figurent dans les modèles prédictifs utilisés dans ce mémoire.

Le tableau ci-dessous présente les métriques et leurs définitions. Sachant que :

- *Parents (c)*: est l'ensemble des classes parents de la classe *c*,
- *Children (c)*: est l'ensemble des classes fils de la classe *c*,
- *Ancestors (c)* : est l'ensemble des classes ancêtres de la classes *c*,
- *Descendants (c)* : est l'ensemble des classes descendants de la classes *c*,
- *Others (c)* = {classes du système} − (*Ancestors (c)* ∪ *Descendants (c)* ∪ *{c}*)
- *M (c)*: est l'ensemble des méthodes de la classe *c*,
- M_{INH} *(c)* : est l'ensemble des méthodes héritées par *c*,
- M_{OVER} *(c)* : est l'ensemble des méthodes redéfinies dans *c*,
- M_{NEW} *(c)*: est l'ensemble des méthodes de *c* ni héritées ni redéfinies,
- M_{PUB} *(c)*, M_{PRIV} *(c)*, M_{PRO} *(c)*, M_{ABS} *(c)* : désignent respectivement l'ensemble des méthodes publiques, privés, protégés et abstraites dans la classe *c*,

[9] La stabilité désigne la stabilité de l'interface d'une classe. On s'intéresse donc à la disparition d'attributs ou de méthodes, celles-ci étant définies par leur signature (nom + type de retour + paramètres).

- $A_I(c)$: est l'ensemble des attributs implémentés dans la classe c,
- $T(a)$: désigne le type de l'attribut a,
- $Par(m)$: est l'ensemble de paramètres de la méthode m.

Symbole	Nom	Définition
DIT	Profondeur de l'arbre d'héritage. *Depth of Inheritance Tree.*	$DIT(c) = \begin{cases} 0, if\ Parents(c) = \emptyset \\ 1 + \max\{DIT(c') : c' \in Parents(c)\}, else. \end{cases}$
NOC	Nombre d'enfants. *Number Of Children.*	$NOC(c) = \lvert Children(c) \rvert$
NMI	Nombre de méthodes héritées. *Number of Methods Inherited.*	$NMI(c) = \lvert M_{INH}(c) \rvert$
NMA	Nombre de méthodes ajoutées. *Number of Methods Added.*	$NMA(c) = \lvert M_{NEW}(c) \rvert$
NMO	Nombre de méthodes redéfinies. *Number of Methods Overridden.*	$NMO(c) = \lvert M_{OVER}(c) \rvert$
CIS	Nombre de méthodes publiques. *Class Interface Size.*	$CIS(c) = \lvert M_{PUB}(c) \rvert$
NAM	La somme de méthodes définies et des méthodes héritées. *Number of Available Methods.*	$NAM(c) = NMA(c) + NMI(c)$
OAM	Ratio de méthodes publiques par le nombre total de méthodes définis. *Operation Access Metric.*	$OAM(c) = \frac{CIS(c)}{NMA(c)}$
NPM	La moyenne des paramètres par méthodes. *Average of the number of parameters per method.*	$NPM(c) = \frac{\sum_{m \in NMA(c)} \lvert \{p \mid p \in Par(m)\} \rvert}{NMA(c)}$

NOP	Nombre de méthodes polymorphes. *Number of polymorphics methods.*	$NOP(c) = \lvert M_{PRO}(c) \rvert + \lvert M_{PUB}(c) \rvert + \lvert M_{ABS}(c) \rvert$
DAC	Couplage d'abstraction de données. *Data Abstraction Coupling.*	$DAC(c) = \lvert \{a \mid a \in A_I(c) \wedge T(a) \in C\} \rvert$
DAC'		$DAC'(c) = \lvert \{T(a) \mid a \in A_I(c) \wedge T(a) \in C\} \rvert$
OCAIC	Couplage classe-attribut importé des autres classes. *Others Class-Attribute Imported Coupling.*	$ACAIC(c) = \lvert \{a \mid a \in A_I(c) \wedge T(a) \in Others(c)\} \rvert$
OCAEC	Couplage classe-attribut exporté vers les autres classes. *Others Class-Attribute Exported Coupling.*	$OCAEC(c) = \sum_{c' \in Others(c)} \lvert \{a \mid a \in A_I(c') \wedge T(a) = c\} \rvert$
OCMIC	Couplage classe-méthode importé des autres classes. *Others Class-Method Imported Coupling.*	$ACMIC(c) = \sum_{m \in M_{NEW}(c)} \lvert \{a \mid a \in Par(m) \wedge T(a) \in Others(c)\} \rvert$
OCMEC	Couplage classe-méthode exporté vers les autres classes. *Others Class-Method Exported Coupling.*	$OCMEC(c) = \sum_{c' \in Others(c)} \sum_{m \in M_{NEW}(c')} \lvert \{a \mid a \in Par(m) \wedge T(a) = c\} \rvert$
NTA	Nombre total d'attributs définis. *Number of Total defined Attributes.*	$NTA(c) = \lvert A_{NEW}(c) \rvert$
NPA	Nombre d'attributs publics. *Number of Public Attributes.*	$NPA(c) = \lvert A_{PUB}(c) \rvert$
NAI	Nombre d'attributs hérités. *Number of attributes inherited.*	$NAI(c) = \lvert A_{INH}(c) \rvert$

NAA	La somme d'attributs définis et d'attributs hérités. *Number of Available Attributes.*	$NAA(c) = NTA(c) + NAI(c)$
DAM	Ratio d'attributs privés et protégés par le nombre total d'attributs définis. *Data Access Metric.*	$DAM(c) = \left\| \frac{\|A_{PRI}(c)\| + \|A_{PRO}(c)\|}{NTA(c)} \right\|$

Tableau 3. Définitions de métriques utilisées dans ce mémoire

3.2 DÉFINITION D'UNE RESTRUCTURATION

Une restructuration (ou "*refactoring* ") est un processus de modification du code source écrit dans un langage à objets, dont l'intention est d'améliorer sa structure interne et d'augmenter sa réutilisabilité, sans modifier le comportement externe du système lorsque certaines préconditions sont vérifiées [FOW99].

Ces transformations permettent de faire évoluer une application en la rendant plus lisible et surtout davantage réutilisable. Un *refactoring* peut donc être vu comme une opération prenant en entrée un code implantant certaines fonctions, avec un niveau de qualité donné (lisibilité, réutilisabilité, modularité,...) et ayant pour résultat un nouveau code implantant les mêmes fonctions mais avec un niveau de qualité supérieure. L'impact du *refactoring* est extrêmement important, car il permet de « revisiter » entièrement une application et de l'améliorer automatiquement d'un point de vue non fonctionnel.

On distingue deux types de restructurations : les restructurations de bas niveau et les restructurations de haut niveau.

3.2.1 Les restructurations de bas niveau

Une restructuration de bas niveau est une modification élémentaire qui peut être appliquée à une classe et qui ne modifie aucunement le comportement de cette classe, lorsqu'un certain nombre de préconditions est vérifié. Par exemple, pour la restructuration

«changement du nom d'une variable », la précondition est qu'il n'y aura pas de collision avec une autre variable de même nom dans la classe.

Les restructurations de bas niveau forment une sorte de catalogue qui sert à décomposer les restructurations de haut niveau, sans changer le comportement du système ni introduire d'erreurs dans le programme. Les tableaux ci-dessous résument la plupart des restructurations de bas niveaux définis dans [OPD92].

Restructuration	Description	Préconditions
Création d'une classe	Définit une nouvelle classe vide.	➢ Pas de classe existante avec le même nom.
Création d'un attribut	Ajout d'un attribut non référencé.	➢ Pas de variable de même nom dans le champ de visibilité de la classe.
Création d'une méthode	Ajout d'une méthode non référencée ou même signature qu'une méthode héritée.	➢ Non existante dans la classe. ➢ Si héritée, non référencée ou équivalente.
Suppression d'une classe	Supprime une classe non référencée.	➢ N'est pas référencée et n'a pas de sous-classe.
Suppression d'un attribut	Supprime un attribut non référencé.	➢ L'attribut ne doit pas être référencé.
Suppression de méthodes	Suppression d'un ensemble de méthodes d'une même classe.	➢ Les méthodes sont redondantes ou non référencées ou référencées par des méthodes à supprimer.
Changement du nom d'une classe	Change le nom de la classe, répercute le changement dans tout le programme.	➢ Pas de classe existante avec le nouveau nom.
Changement du nom d'un attribut (ou variable)	Change le nom d'une variable, répercute le changement à l'intérieur de son champ de visibilité.	➢ Pas de collision avec une autre variable de même nom.

Changement du nom d'une méthode	Change le nom d'une méthode et répercute le changement pour les sous-classes et les références.	➢ Pas de méthode avec la même signature dans la classe. ➢ Si publique, pas de méthode d'une sous-classe avec le même nom. ➢ Si signature identique à celle d'une méthode héritée, elles doivent être équivalentes.
Changement du type	Change le type d'un ensemble de variables et de méthodes.	➢ Préserve la validité des affectations.

Tableau 4. Les restructurations concernant la création, suppression et changement d'une entité

Restructuration	Description	Préconditions
Changement du mode d'accès	Change le mode d'accès à un attribut ou une méthode.	➢ Si le nouveau mode est privé, le membre doit être référencé à l'interne uniquement. ➢ Pour devenir protégé ou public, un attribut ne doit pas être défini dans une sous-classe. ➢ Si l'ancien mode était privé et si le membre est une méthode, s'il existe une méthode héritée avec la même signature, celle-ci n'est pas référencée dans la classe ou les sous-classes. ➢ Si l'ancien mode était protégé et le nouvel est public, le membre ne doit être référencé que de manière interne dans la classe et les sous-classes.

Tableau 5. La restructuration concernant le changement du mode d'accès

Restructuration	Description	Préconditions
Ajout du corps d'une méthode	Ajoute du code dans une méthode vide et non référencé d'une classe.	➢ Le code ajouter doit se compiler, sans erreur.
Suppression du corps d'une méthode	Efface le corps d'une méthode.	➢ La méthode n'est pas référencée.

Tableau 6. Les restructurations concernant l'ajout et la suppression du corps d'une méthode

Restructuration	Description	Préconditions
Ajout d'un paramètre à une méthode	Ajoute d'un paramètre à la définition d'une méthode. Crée dynamiquement une instance du nouveau paramètre pour chaque appel de la méthode.	➢ Pas de collision avec une autre variable dans le champ de visibilité de la méthode. ➢ Pas de collision avec d'autres signatures. ➢ Valeur par défaut visible par toutes les références.
Suppression d'un paramètre d'une méthode	Supprime un paramètre de la définition d'une méthode et si elle n'est pas privée, supprime le paramètre de toutes les sous-classes ou la méthode est redéfinie.	➢ Le paramètre n'est pas référencé dans la méthode ni dans les méthodes des sous-classes qui la redéfinissent. ➢ La méthode contenant le paramètre n'est pas définie dans une super-classe de la classe considérée. ➢ Les expressions affectées au paramètre n'ont pas d'effet de bord.

Tableau 7. Les restructurations concernant l'ajout et la suppression d'un paramètre d'une méthode

Restructuration	Description	Préconditions
Conversion des références à une variable en appels à une méthode	Convertit toutes les références à une variable en appels à une méthode d'accès à la variable.	➢ Les méthodes d'accès et de mise à jour ont déjà été définies.

Tableau 8. La restructuration concernant le remplacement des références à un attribut par un appel à une méthode d'accès

Restructuration	Description	Préconditions
Remplacement d'un bloc de code par un appel de méthode	Remplace une suite d'instructions par un appel à une méthode.	➢ La méthode est visible depuis l'emplacement ou le code sera enlevé. ➢ L'appel de méthode est équivalent au code qui sera remplacé.
Remplacement de l'appel à une méthode par un bloc de code	Remplace un appel à une méthode par le corps de la méthode.	➢ Tous les membres référencés par la méthode appelée sont accessibles depuis la méthode appelante.

Tableau 9. Les restructurations concernant le remplacement d'un segment de code par un appel à une méthode et inversement

Restructuration	Description	Préconditions
Changement de super-classe	Remplace la super-classe d'une classe.	➢ Préserve la validité des affectations. ➢ Tous les membres hérités seront hérités également de la nouvelle super-classe. ➢ Pas de collision de membres avec la nouvelle super-classe ou équivalence avec les membres hérités. ➢ Les constructeurs sont équivalents dans l'ancienne et la nouvelle super-classe.

Tableau 10. La restructuration concernant le changement de super-classe

Restructuration	Description	Préconditions
Déplacement d'un attribut vers une super-classe	Supprime un attribut des classes ou il est défini et l'ajoute à une super-classe.	➢ Dans les sous-classes où il est défini, l'attribut est défini de la même façon. ➢ L'attribut n'est pas déjà défini localement dans la super-classe.
Déplacement d'un attribut vers des sous-classes	Déplace un attribut d'une classe vers chacun de ses descendants directs.	➢ L'attribut n'est pas référencé par la classe ou par des instances de la classe. ➢ L'attribut est déjà hérité par les sous-classes.

Tableau 11. La restructuration concernant le déplacement d'un attribut

3.2.2 Les restructurations de haut niveau

Une restructuration de haut niveau est une modification majeure de la conception du système qui ne modifie aucunement son comportement externe. Elle consiste, par exemple, en la décomposition d'une classe en un ensemble de sous-classes (spécialisation) ou au changement de la relation d'héritage entre deux classes en une relation d'agrégation.

Une transformation de haut niveau peut être décomposée en un ensemble fini de restructurations de bas niveau. Cette opération préservera le fonctionnement initial du programme puisque les restructurations de bas niveau qui la composent préservent, par définition, le comportement du programme [OPD92]. L'impact d'une restructuration de haut niveau sur les métriques est obtenu en cumulant l'impact de restructurations de bas niveau sur ces métriques.

Dans ce qui suit est une brève description de quatre restructurations de haut niveau, tirées de [OPD92] et [MIC99b], utilisées dans ce mémoire. La mesure de l'impact de ces restructurations sur les valeurs de métriques sera présentée dans la section 3.3.

3.2.2.1 La restructuration « Généralisation »

Cette restructuration consiste à créer une classe abstraite à partir d'un ensemble de classes possédant une même super-classe. Elle est composée des restructurations élémentaires suivantes :

- *création d'une classe,*
- *changement de super-classe,*
- *ajout d'un paramètre à une méthode,*
- *création d'une méthode,*
- *remplacement d'un segment de code par un appel de méthode.*

Exemple : dans un système de gestion d'une institution d'enseignement, on trouve les classes Étudiant, Professeur, Administrateur et Personne. Au début, les trois premières classes possèdent la classe Personne comme super-classe (cf. figure 4 ci-dessous), étant donnée l'existence de plusieurs comportements communs entre elles. Or, les classes Professeur et Administrateur sont toutes les deux des employés de l'institution, et renferment plusieurs attributs et méthodes similaires. D'où l'utilisation de la restructuration « Généralisation » pour créer la classe Employé, qui sera la super-classe de Professeur et Administrateur et hérite de la classe Personne.

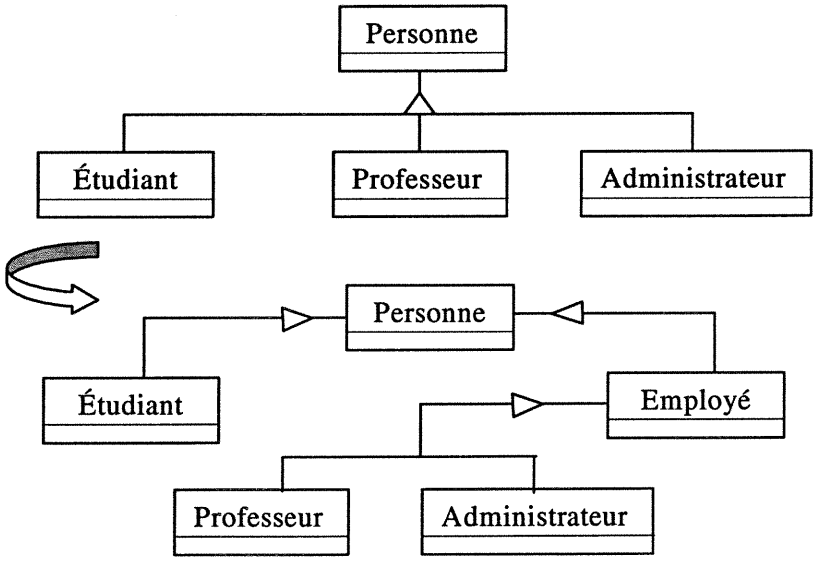


Figure 4. La restructuration Généralisation

3.2.2.2 La restructuration « Spécialisation »

Cette transformation consiste à créer plusieurs sous-classes à partir d'une classe donnée. Elle est composée des restructurations élémentaires suivantes :

- *création d'une classe,*
- *création d'une méthode,*
- *remplacement d'un segment de code par un appel de méthode,*
- *changement du mode de contrôle d'accès.*

Exemple : pour schématiser la relation qui existe entre les classes Facteur et Personne, la classe Courrier a été créée (cf. figure 5 ci-dessous). Cette classe, tel qu'elle est, ne distingue pas entre un courrier lettre d'un autre colis. La solution consiste à spécialiser la classe Courrier en deux sous-classes, Lettre et Colis, permettant ainsi la représentation de chaque type de courrier. La restructuration « Spécialisation » est utilisée pour spécialiser une telle classe.

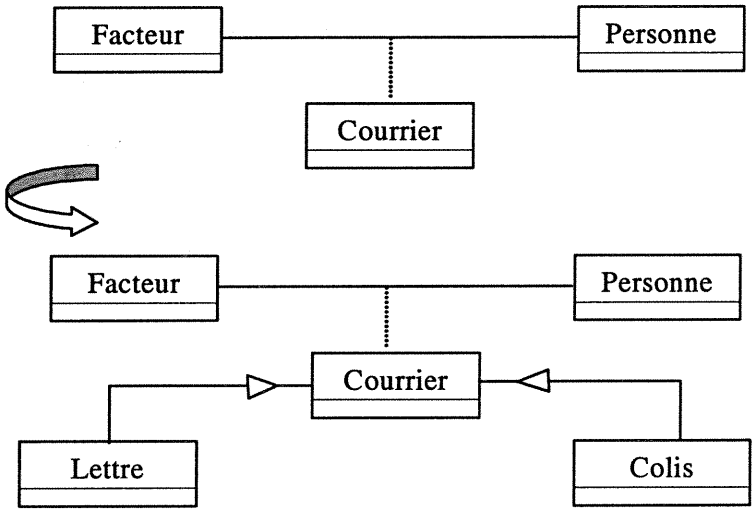


Figure 5. La restructuration Spécialisation

3.2.2.3 La restructuration « Composition »

Cette opération permet d'extraire des éléments d'une classe pour former une ou plusieurs nouvelles classes qui deviendront des agrégats de la classe initiale. Cette restructuration peut être vue comme l'ensemble des restructurations élémentaires suivantes :

- *création d'une classe,*
- *création d'un attribut,*
- *création d'une méthode,*
- *changement de référence à une méthode,*
- *supprimer un attribut,*
- *supprimer une méthode.*

Exemple : considérons la classe Mail qui modélise un courrier électronique (ou "*email*"), dont ses attributs sont illustrés dans la figure 6 ci-dessous. Chaque attribut de cette classe est un attribut complexe, et il sera mieux de le présenter sous forme d'une classe indépendante. La restructuration « Composition » est utilisée pour extraire les quatre classes de la classe Mail. Les classes créées seront agrégées à la classe agrégat Mail.

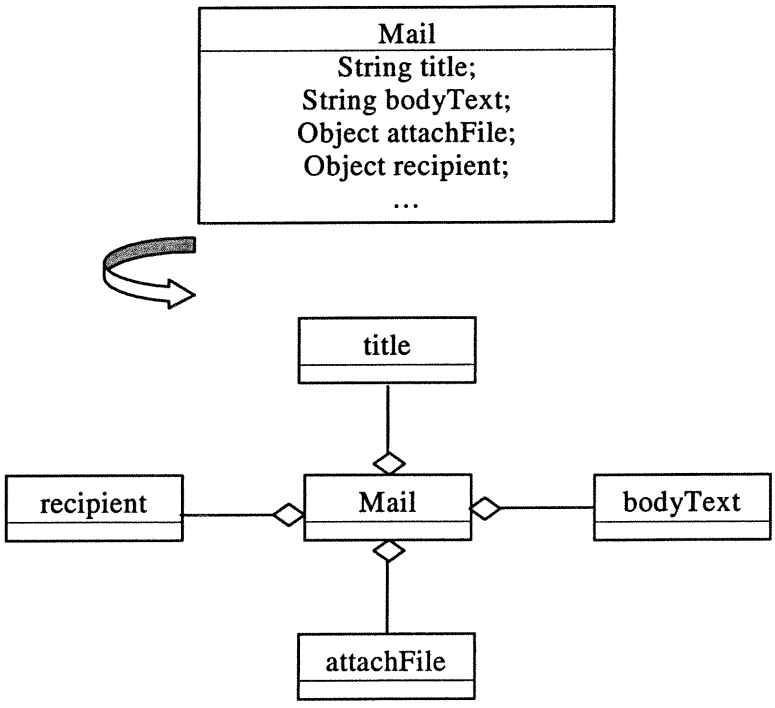


Figure 6. La restructuration Composition

3.2.2.4 La restructuration « Conversion d'une relation d'héritage »

Cette restructuration transforme une association, modélisée sous forme d'héritage, en une agrégation. Elle est composée des restructurations élémentaires suivantes :

- *création d'un attribut,*
- *création d'une méthode,*
- *changement du mode d'accès,*
- *changement de super-classe.*

Exemple : la figure 7 ci-dessous présente l'exemple d'une relation d'héritage mal conçue, qui existe entre les deux classes Voiture et Moteur. En effet, le concepteur tend d'hériter la classe Voiture de la classe Moteur, en disant qu'une voiture n'est autre qu'un moteur avec quelque attributs et méthodes supplémentaires. Après une bonne réflexion, le concepteur réalise qu'une voiture est composée, entre autres, d'un moteur, et la relation d'héritage dû être une agrégation, dans laquelle la classe sous-classe aurait été la classe agrégat. La restructuration « Conversion d'une relation d'héritage » est utile dans ce cas.

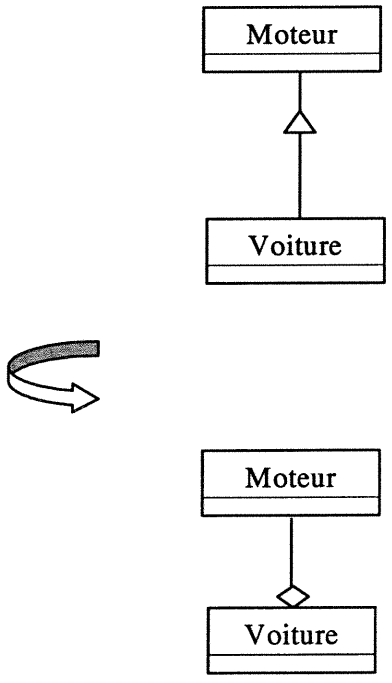


Figure 7. La restructuration Conversion d'une relation d'héritage en une agrégation

3.3 INFLUENCE DES RESTRUCTURATIONS SUR LES MÉTRIQUES

3.3.1 Mesure de l'impact des restructurations de bas niveau

Dans les sections qui suivent, nous allons évaluer l'effet produit par les restructurations de bas niveau sur les valeurs de métriques, utilisées dans les modèles prédictifs. Cette mesure est restreinte aux restructurations élémentaires figurant dans les quatre transformations de haut niveau utilisées dans la partie expérimentation. Elle s'est inspirée de celle présentée dans [MIC99a][10].

Tout d'abord, pour chaque restructuration, on cherche les classes et les métriques impliquées dans cette modification élémentaire. Ensuite, la variation de la valeur d'une métrique est calculée selon le contexte (ou les conditions) dans lequel la variation est valide. Par exemple, considérons la restructuration « création d'une méthode » qui ajoute une méthode *m* à la classe *c*. Les classes impliquées dans cette opération sont *c*, ses descendants ainsi que les classes qui sont de même type que la méthode ajoutée. Cette restructuration affecte, entre autres, le nombre de méthodes ajoutées (*NMA*) de *c*. La valeur de *NMA* augmente d'un point si la méthode *m* n'est pas déjà héritée.

Dans les tableaux qui suivent, nous désignons par :

- **Par (*m*)** : le nombre de paramètres de la méthode ***m*** ;
- **ParTC (*m*)** : le nombre de paramètres de ***m*** qui sont de type classe ;
- **M_{PRIV} (*c*)** et **M_{PRO} (*c*)**: le nombre de méthodes privées et protégées de la classe ***c*** ;
- **SMI (*s*)** : l'ensemble d'appels statiques de segment du code ***s*** ;
- **NSI (*m, m'*)** : le nombre d'appels statiques de la méthode ***m'*** depuis ***m*** ;
- **NumCM (*c, c'*)** : le nombre de méthodes communes entre les classes ***c*** et ***c'***.

Une colonne intitulée « Variation de valeurs de métriques » et qui contient, par exemple, DIT + (NAA +1), se lit comme suit : la valeur de métrique DIT après la restructuration est augmente de la valeur de (NAA +1).

---

[10] Miceli a évalué l'impact des restructurations sur les métriques d'héritage et de couplages.

3.3.1.1 Création d'une classe

Cette restructuration crée une classe en tant que nouvelle feuille de la hiérarchie de classes, comme le montre la figure 8 ci-dessous. *c'* désigne la classe nouvellement créée et *c''* la super-classe de *c'*.

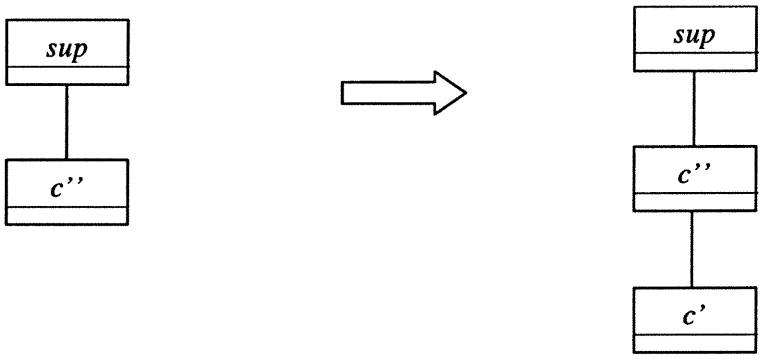


Figure 8. La création d'une classe

Le tableau suivant résume les variations de valeurs de métriques pour les classes impliquées dans cette restructuration.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
c'	-	✓ DIT + [DIT (*c''*) + 1] ✓ NAA + NAI ✓ NAM + NMI
c''	-	✓ NOC + 1

Tableau 12. Impact de la création d'une classe sur les métriques

3.3.1.2 Création d'un attribut

Cette restructuration insère un attribut *a* dans une classe *c'*. Cette variable est créée de manière à éviter la collision avec les autres attributs, qui portent le même nom.
Le tableau suivant résume l'impact de cette opération sur les métriques.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
Descendante de c'	-	✓ NAA + 1

Type de l'attribut ajouté	-	✓ OCAEC + 1
c'	-	✓ DAC + 1 ✓ DAC' + 1 ✓ OCAIC + 1
c'	*a est privé/protégé*	✓ NAA + 1 ✓ DAM + [NPA/[(NTA) × (NTA+1)]
c'	*a est public*	✓ NPA + 1 ✓ NAA + 1 ✓ DAM – [DAM/(NTA+1)]

Tableau 13. Impact de la création d'un attribut sur les métriques

3.3.1.3 Création d'une méthode

Cette restructuration insère une méthode *m* dans une classe *c'*. Cette méthode ne doit pas exister dans la classe et si elle est héritée, elle est équivalente à la méthode déjà héritée. Le tableau ci-dessous résume l'impact de cette restructuration sur les métriques.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
Descendants de c'	- *m n'existe pas dans les classes descendantes* - *m est protégée/public*	✓ NMI +1 ✓ NAM + 1
Descendants de c'	- *m existe déjà dans les classes descendantes* - *m est protégée/public*	✓ NMO + 1 ✓ NMA – 1 ✓ NAM + 1
Type des paramètres de m	-	✓ OCMEC + |ParTC (*m*)|
c'	*m est héritée*	✓ NMI + 1 ✓ NMO + 1 ✓ OCMIC + |Par (*m*)|
c'	- *m est non héritée* - *m est privée/protégée*	✓ NMA + 1 ✓ NAM + 1 ✓ OAM – [OAM/(NMA + 1)] ✓ NPM – [NPM/(NMA + 1)] ✓ OCMIC + |Par (*m*)|
c'	*m est public*	✓ CIS + 1 ✓ NAM + 1 ✓ OAM + (($\|M_{PRI}(c')\|$ + $\|M_{PRO}(c')\|$) / (NMA × (NMA + 1))) ✓ NPM – [NPM/(NMA + 1)]

Tableau 14. Impact de la création d'une méthode sur les métriques

3.3.1.4 Suppression d'un attribut

Cette restructuration supprime un attribut *a*, non référencé, d'une classe *c'*. L'impact de ce changement est donné par le tableau suivant.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs des métriques
Descendants de c'	*a est protégé/public*	✓ NAA – 1
Type de l'attribut supprimé	-	✓ OCAEC – 1
c'	*a est de type classe*	✓ DAC – 1 ✓ DAC' – 1 ✓ OCAIC – 1
c'	*a est privé/protégé*	✓ NAA – 1 ✓ DAM – [NPA/((NTA) × (NTA – 1))]
c'	*a est public*	✓ NPA – 1 ✓ NAA – 1 ✓ DAM + DAM/(NTA – 1)

Tableau 15. Impact de la suppression d'un attribut sur les métriques

3.3.1.5 Suppression d'une méthode

La suppression d'une méthode *m* d'une classe *c'* est effectuée lorsqu'elle n'est plus référencée. Le tableau ci-dessous regroupe les changements des valeurs des métriques pour cette opération.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
Descendants de c'	- *m n'existe pas dans les classes descendantes* - *m est protégée / public*	✓ NMI –1 ✓ NAM – 1
Descendants de c'	- *m existe déjà dans les classes descendantes* - *m est protégée / public*	✓ NMO – 1 ✓ NMA + 1 ✓ NAM – 1
Type des paramètres de m	-	✓ OCMEC – |ParTC (*m*)|

c'	*m est privée/protégée*	✓ NAM – 1 ✓ OAM + [OAM/(NMA–1)] ✓ NPM + [NPM/(NMA – 1)] ✓ NMA – 1 ✓ OCMIC – |Par (*m*)|
c'	*m est public*	✓ CIS – 1 ✓ NAM – 1 ✓ OAM – (($\|P_{RIV}(c')\| + \|P_{ROT}(c')\|$)/ [(NMA x (NMA – 1)))] ✓ NPM + NPM/ (NMA – 1) ✓ NMA – 1 ✓ OCMIC – |Par (m)|

Tableau 16. Impact de la suppression d'une méthode sur les métriques

3.3.1.6 Ajout d'un paramètre à une méthode

Dans cette restructuration, on ajoute un paramètre *P* à une méthode *m* d'une classe *c'*. L'impact de cette opération sur les valeurs des métriques est donné par le tableau ci-dessous.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
Type de P	-	✓ OCMEC + 1
c'	*P est de type classe*	✓ OCMIC + 1 ✓ NPM + [(1/NMA)]

Tableau 17. Impact de l'ajout d'un attribut à une méthode sur les métriques

3.3.1.7 Remplacement d'un segment de code par un appel de méthode

Cette opération supprime un segment de code *s* d'une méthode *m,* appartenant à une classe *c'*, et le remplace par un appel à une méthode m_c d'une classe *c*.

Le tableau suivant regroupe l'influence de cette restructuration sur les valeurs des métriques.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
c'	-	✓ OCMIC – $\sum_{m' \in SIM(s)} NSI(m,m')$

Tableau 18. Impact du remplacement d'un segment de code par un appel de méthode sur les métriques

3.3.1.8 Changement de super-classe

Cette restructuration change la super-classe d'une classe. Soit c_d la classe déplacée, c_a l'ancienne super-classe de c_d et c_n, la nouvelle classe. Le tableau suivant résume l'impact de changement sur les valeurs des métriques.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
Type des attributs de c_d	*les classes sont des ancêtres de* c_n	✓ OCAEC – $\|A_I(c_d)\|$ ✓ OCMEC – $\|A_I(c_d)\|$
Type des attributs de c_d	*les classes sont des ancêtres de* c_d	✓ OCAEC + $\|A_I(c_d)\|$
Type de paramètres des méthodes de c_d	-	✓ OCMEC + $\sum_{m \in M(cd)} Par(m)$
c_d	-	✓ DIT (c_n) +1 ✓ NMI + [0, NMA (c_n)] ✓ NMO + $NumC_M(c_d, c_n)$ ✓ NMA – $NumC_M(c_d, c_n)$ ✓ NAA + NAI ✓ NAM + NMI
c_d	*les types des attributs et des paramètres sont des ancêtres de* c_d	✓ OCAIC + [0, NTA] ✓ OCMIC + $\sum_{m \in M(cd)} Par(m)$
c_d	*le type des attributs est* c_n *ou sont des ancêtres de* c_n	✓ OCAIC – $\|A_I(c_d)\|$
c_d	*le type des paramètres est* c_n *ou sont des ancêtres de* c_n	✓ OCMIC – $\sum_{m \in M(cd)} Par(m)$
c_a	-	✓ NOC – 1
c_n	*les attributs des* c_d *ont* c_n *comme type*	✓ OCAEC – $\|A_I(c_d)\|$ ✓ OCMEC – $\|A_I(c_d)\|$

Tableau 19. Impact du changement de super-classe sur les métriques

3.3.1.9 Changement du mode de contrôle d'accès d'un attribut/méthode

Cette restructuration modifie le mode de contrôle d'accès (public, protégé, privé) d'un attribut ou d'une variable. Soit *a* l'attribut et *m* la méthode d'une classe *c'* dont on va modifier leur mode d'accès.

Le tableau suivant résume l'impact de l'opération courante sur les métriques.

Rôle de la classe dans la restructuration	Conditions	Variation de valeurs de métriques
Descendantes c'	*L'attribut/méthode est privé*	✓ NAA + 1 ✓ NMI + 1 ✓ NMO + 1 ✓ NMA – 1 ✓ NAM + 1
Descendantes c'	*L'attribut/méthode est non privé*	✓ NAA – 1 ✓ NMI – 1 ✓ NMO –1 ✓ NMA +1 ✓ NAM – 1
c'	*L'attribut/méthode passe de privé/protégé à public*	✓ NPA + 1 ✓ DAM – (1/NTA) ✓ CIS + 1 ✓ OAM + (1/NMA)
c'	*L'attribut/méthode passe de public à privé/protégé*	✓ NPA – 1 ✓ DAM + (1/NTA) ✓ CIS – 1

Tableau 20. Impact de changement du mode de contrôle d'accès d'un attribut ou d'une méthode sur les métriques

3.3.2 Mesure de l'impact des restructurations de haut niveau

3.3.2.1 Création d'une classe abstraite

Cette opération consiste à créer une classe abstraite à partir d'un ensemble de classes possédant une même super-classe. La classe créée inclura les comportements communs de ces classes.

Soit c_1, c_2, c_3,..., c_N N classes à factoriser, c_s leur super-classe initiale et c_a la classe abstraite à créer. Les figures 9 et 10 montrent la hiérarchie de classes avant et après cette transformation.

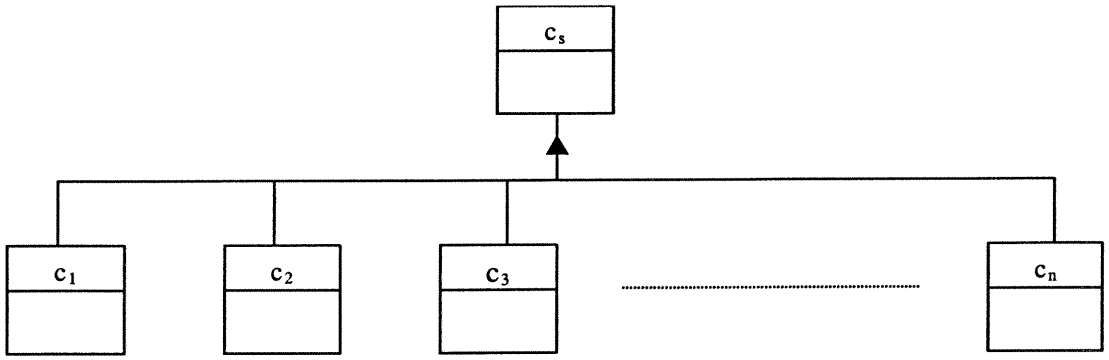


Figure 9. Hiérarchie de classes avant la transformation

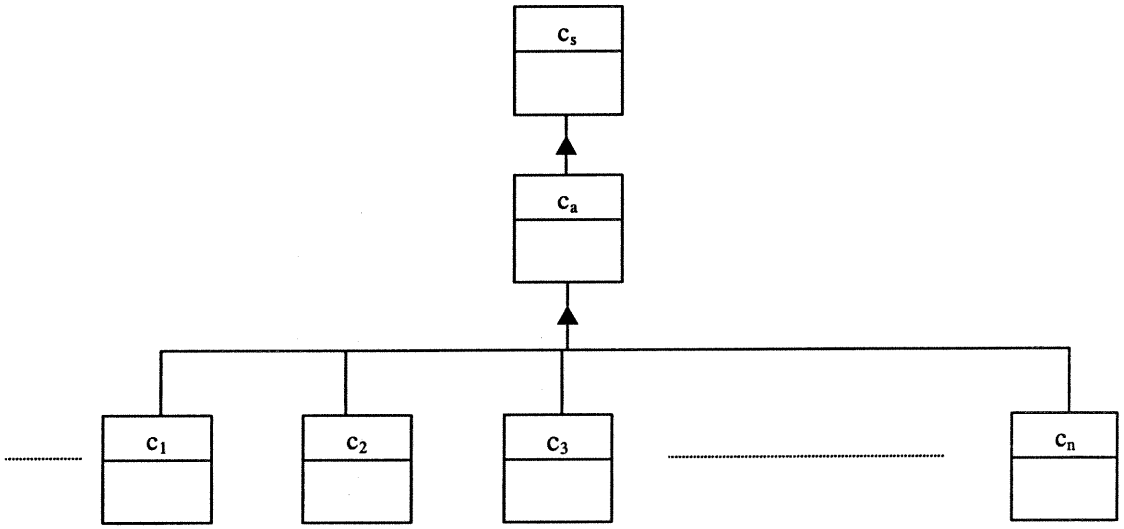


Figure 10. Hiérarchie de classes après la transformation

La transformation actuelle se décompose en une série des restructurations élémentaires suivantes [OPD92] :

- création d'une classe : cette opération est utilisée pour créer la classe c_a en tant qu'un fils de la classe c_s.
- changement de super-classe : cette opération est utilisée pour déplacer les classes $c_{i\ (i=1,\ ...N)}$ sous la classe c_a, nouvellement créée. La classe c_a deviendra ainsi la super-classe de c_i.

- ajout d'un paramètre à une méthode : avant d'ajouter les signatures des méthodes communes aux classes c_i dans la classe abstraite, celles-ci devront avoir une même signature. L'ajout d'un paramètre à une méthode pourra être utilisé à cette fin. P_A représente l'ensemble de paramètres ajoutés aux méthodes.
- création d'une méthode : cette opération est appliquée deux fois à la classe abstraite c_a. D'une part, pour ajouter les signatures des méthodes à abstraire qui sont communes aux classes c_i et, d'autre part, pour créer des méthodes équivalentes aux segments de code communs entre les sous-classes c_i. M_A et M_C correspondent respectivement à l'ensemble de méthodes ajoutées et créées.
- remplacement d'un segment de code par un appel de méthode : dans chaque classe c_i, on remplace le segment de code par un appel à une méthode convenable, récemment créée dans la super-classe.

Comme nous l'avons mentionné préalablement, la mesure de l'impact de cette transformation sur les métriques est obtenue en cumulant l'impact des restructurations de bas niveau qui la composent. Les tableaux ci-dessous regroupent les variations de valeurs de métriques pour chaque classe ou catégorie de classes.

Métrique	Variation de valeur
DIT	+1
NMA	+ [– |M_A|, 0]
NMO	+ [0, |M_A|]
NMI	+ |M_C|
NAA	+ NAI
NAM	+ NMI
NPM	+ [1 / NMA]
OCMIC	+ [0, |P_A|]

Tableau 21. Impact de la création d'une classe abstraite sur les métriques des classes factorisées

Métrique	Variation de valeur
NOC	+ (1– N)

Tableau 22. Impact de la création d'une classe abstraite sur les métriques de super-classe initiale

Métrique	Variation de valeur
OCMEC	$+\left[0, \sum_{m_i \in MA} Par(m_i)\right]$

Tableau 23. Impact de la création d'une classe abstraite sur les métriques des classes utilisées dans la signature des méthodes créées à partir de codes communs

Métrique	Variation de valeur
OCMEC	+ [0, $\|P_A\|$]

Tableau 24. Impact de la création d'une classe abstraite sur les métriques de classes utilisées comme type des paramètres ajoutés aux méthodes abstraites

Métrique	Variation de valeur
DIT	DIT (c_s) +1
NOC	N
NMI	[0, NAM (c_s)]
NMO	[0, NAM (c_s)]
NMA	[0, $\|M_A\|$] + $\|M_C\|$
OCMIC	$\left[\sum_{m_i \in MA} \|Par(m_i)\| + \sum_{m_i \in Mc} \|Par(m_i)\|\right]$

Tableau 25. Impact de la création d'une classe abstraite sur les métriques de nouvelle super-classe

3.3.2.2 Création de sous-classes

L'amélioration de la conception d'un système orienté objet passe parfois par la décomposition des classes larges et complexes en plusieurs sous-classes plus concrètes. D'une façon générale, une classe complexe incorpore une abstraction générale et plusieurs cas concrets, qui sont candidats à un processus de spécialisation ultérieures.

La transformation actuelle crée de nouvelles sous-classes d'une classe qui est initialement une feuille dans la hiérarchie de classes. Les sous-classes correspondent aux conditions d'une expression conditionnelle. Pour chaque condition, une sous-classe (qui représente un sous-type) est créée dont l'abstraction de conception implique cette condition.

La transformation se décompose en étapes comme suit :

i- Sélection des expressions conditionnelles dont les conditions suggèrent des sous-classes ;

ii- Pour chaque condition, créer une sous-classe avec une invariante équivalente à la condition ;

iii- Pour chaque expression conditionnelle, créer une méthode dans chaque sous-classe. Simplifier et spécialiser le corps des méthodes pour chaque sous-classe, selon son invariante ;

iv- Spécialiser plusieurs ou toutes les expressions qui créent des instances de super-classe. Spécialiser une expression implique le remplacement d'une expression qui crée une instance de super-classe par une autre expression qui crée une instance de l'une des sous-classes créées.

Les figures ci-dessous montrent l'état de la classe *MemClass* avant (figure 11) et après (figure 12) l'opération.

```
class MemClass {

        int i ;
        .....

        void selection (int val1, int val2, int val3) {
                if (i = = val1)
                        < Treatment1>

                else if (i = = val2)
                        < Treatment2>

                else if (i = = val3)
                        < Treatment3>
        }
}
```

Figure 11. La classe *MemClass* avant la transformation

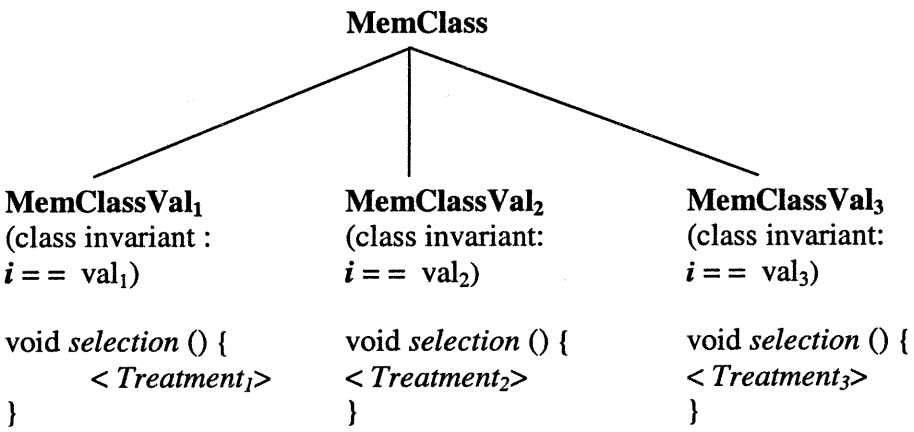


Figure 12 : La classe *MemClass* spécialisée après la restructuration

Soit c_s, la classe à spécialiser.

La transformation actuelle peut être considérée comme l'ensemble des restructurations élémentaires suivantes :

- création d'une classe: pour chaque valeur de la condition, on crée une sous-classe de c_s;
- création d'une méthode : chaque segment de code correspondant à une expression conditionnelle est transformé en une méthode de c_s. Cette restructuration est également utilisée pour ajouter ces méthodes aux sous-classes ;
- remplacement d'un segment de code par un appel de méthode : chaque segment de code, transformé en une méthode, est remplacé par un appel à cette méthode ;
- changement du mode de contrôle d'accès : cette opération change le mode d'accès privé d'une variable ou d'une méthode, référencée par les expressions conditionnelles, en un mode protégé.

L'impact de cette restructuration sur les métriques est calculé en cumulant l'impact de ses restructurations de bas niveau sur les valeurs de métriques.

Les tableaux ci-dessous regroupent les variations de valeurs de métriques pour chaque classe ou catégorie de classes. L'ensemble des expressions conditionnelles est représenté par EnsExpCond, dont la cardinalité est égale à M, et le nombre de sous-classes de c_s, créées par la présente transformation, est représenté par N.

Métrique	Variation de valeur
NOC	+ N
NMA	+ M
NAM	+ M
OAM	– [OAM/(NMA + M)]
NPM	– [NPM/(NMA + M)]
OCMIC	$+\left[0, \sum_{m_i \in EnsExpCond} Par(m_i)\right]$

Tableau 26. Impact de la création de sous-classes sur les métriques de la classe à spécialiser

Métrique	Variation de valeur
OCMEC	$+\left[0,(1+N)\sum_{m_i \in EnsExpCond} Par(m_i)\right]$

Tableau 27. Impact de la création de sous-classes sur les métriques des classes utilisées dans les signatures des nouvelles méthodes

Métrique	Variation de valeur
DIT	DIT (c_s) + 1
NMO	+ M
NMI	[0, 2 × M (c_s)]
NAM	+ [NMI + M (c_s)]
NAA	+ NAI
OCMIC	$+\left[0,\sum_{m_i \in EnsExpCond} Par(m_i)\right]$

Tableau 28. Impact de la création de sous-classes sur les métriques des nouvelles sous-classes

3.3.2.3 Création d'un nouveau composant

Cette transformation consiste à regrouper quelques membres publics d'une classe c_a, pour créer une nouvelle classe c_c. Ceci implique le déplacement d'un certain nombre de membres définis dans c_a vers la classe c_c. La nouvelle classe créée deviendra une classe agrégée de la classe c_a.

Une précondition imposée sur cette opération est que les membres déplacés ne soient pas déjà hérités par des sous-classes de c_a.

La création d'un nouveau composant passera par les étapes successives suivantes :

i- Création de la nouvelle classe c_c et déplacement des membres adéquats de c_a vers c_c ;

ii- Création d'une instance de c_c dans c_a ;

iii- Modification des références aux méthodes déplacées de c_a vers c_c ;

iv- Suppression des membres de c_a déplacés vers c_c.

Cette transformation est décomposée en ces restructurations élémentaires :

- création d'une classe : pour créer la classe c_c.
- création d'un attribut : pour copier les attributs déplacés vers c_c, ainsi que pour insérer un attribut (ou une instance) de type c_c dans la classe c_a.
- création d'une méthode : pour copier les méthodes déplacées vers c_c.
- changement de référence à une méthode : les références aux méthodes déplacées de c_a sont remplacées par des références à des méthodes de c_c.
- supprimer un attribut : pour supprimer les attributs de c_a déplacés vers c_c.
- supprimer une méthode : pour supprimer les méthodes de c_a déplacées vers c_c.

L'impact de cette restructuration sur les métriques est présenté dans les tableaux ci-dessous, selon le rôle de la classe dans la transformation.
Dans les tableaux ci-dessous, *EnsATR* représente l'ensemble des attributs transférés ($|EnsATR| = N$) et *EnsMTR*, l'ensemble des méthodes déplacées ($|EnsMTR| = M$).

Métrique	Variation de valeur
DAC	+1 – [0, N]
DAC'	+1 – [0, N]
OCAIC	+1 – [0, N]
OCMIC	$-\left[0, \sum_{m_i \in EnsMTR} Par(m_i)\right]$
NAA	+1 – N
DAM	+ [NPA/(NTA) × (NTA + 1 – N)]

Tableau 29. Impact de la création d'un nouveau composant sur les métriques de la classe initiale

Métrique	Variation de valeur
OCAEC	[0, N]

Tableau 30. Impact de la création d'un nouveau composant sur les métriques des classes utilisées comme type d'attributs transférés

Métrique	Variation de valeur
OCMEC	$+\left[-\sum_{mi \in EnsMTR} Par(m_i), +\sum_{mi \in EnsMTR} Par(m_i)\right]$

Tableau 31. Impact de la création d'un nouveau composant sur les métriques des classes utilisées dans les signatures des méthodes transférées

Métrique	Variation de valeur
DIT	1
NMA	M
DAC	[0, N]
DAC'	[0, N]
OCAIC	[0, N]
NAA	+ N
DAM	+ [NPA/(NTA) × (NTA + N)]
NAM	+ M
OAM	− [OAM/(NMA + M)]
NPM	+ [NPM /(NMA + M)]
OCAEC	1
OCMIC	$+\left[0, \sum_{mi \in EnsMTR} Par(m_i)\right]$

Tableau 32. Impact de la création d'un nouveau composant sur les métriques de la classe nouvellement créée

3.3.2.4 Conversion d'une association, modélisée sous forme d'héritage, en une agrégation

L'héritage n'est pas la seule forme de partage du comportement : les types paramétrés, la composition et l'agrégation sont aussi des moyens de réutilisation. Parfois, une relation d'agrégation n'est évidente qu'après l'étape d'implantation. En effet, durant la phase de conception, un programmeur tente de faire hériter une classe d'une autre parce qu'elles ont des comportements similaires. Dès qu'une version de l'application tourne sur la machine, les relations entre classes deviennent de plus en plus tangibles, ce qui entraîne la révision de plusieurs liens entre elles.
La transformation actuelle supporte l'évolution d'une relation modélisée sous forme d'héritage en une relation d'agrégation. Tous les comportements hérités auparavant par la classe fils seront délégués à une instance de la classe mère, créée dans la classe fils.

Dans la présente transformation, c_s représente la super-classe d'une classe c_a. *EnsAH* et *EnsMH* correspondent respectivement à l'ensemble des attributs et des méthodes héritées par c_a.

La conversion de la relation d'héritage, entre c_s et c_a, en une agrégation se décompose en ces étapes :

i- création d'un attribut de type c_s dans c_a ;

ii- pour chaque élément *a* de *EnsAH* :

   a. création de deux méthodes d'accès à l'attribut *a* dans c_s, soit *get_a* et *set_a* respectivement ;

   b. remplacement des références à cet attribut par un appel à l'une de ses méthodes d'accès appropriées, selon qu'il y ait une affectation ou une utilisation de sa valeur ;

   c. changement du mode d'accès public de *a* en mode protéger.

iii- copier les éléments de *EnsMH*, incluant les méthodes d'accès récemment créés, dans la classe c_a ;

iv- pour chaque élément a de *EnsAH*, remplacer dans ses méthodes d'accès les références à cet attribut par un appel à une méthode convenable de l'instance créée dans c_a ;

v- pour chaque méthode copiée dans la sous-classe c_a, restant de l'étape précédente, remplacer le corps de cette méthode par un appel à la même méthode de l'instance créée dans c_a ;

vi- changement de super-classe de c_a.

Cette transformation pourra être décomposée en une série des restructurations élémentaires comme suit :

- création d'un attribut : pour créer une instance de c_s dans c_a ;
- création d'une méthode : utilisée pour créer les méthodes d'accès *get* et *set* dans c_s. Cette opération est également utilisée pour copier les éléments de *EnsMH* dans c_a;
- changement du mode d'accès : utilisée pour changer le mode public d'accès des attributs hérités de c_s en un mode protégé ;
- changement de super-classe: pour déplacer c_a sous une autre classe c_n.

Les tableaux suivants montrent l'impact de cette transformation sur les valeurs des métriques, par catégorie de classes.

Métrique	Variation de valeur
DIT	+ 1
DAC	+ 1
DAC'	+ 1
NAA	+ [NAI + 1]
DAM	+ [NPA/(NTA) × (NTA+1)]
NMI	+ [0, NMA (c_n)]
NMA	+ [NMI – |NumCM (c_s, c_a)|]
NAM	+ NMI

NMO	+ |NumCM (c_s, c_a)|
OCMIC	+ $\sum_{mi \in M(ca)} \lvert Par(m_i) \rvert$
OCMEC	+ $\sum_{mi \in M(ca)} \lvert Par(m_i) \rvert$
OCAIC	+ [0, NMA (c_n)]
OCMIC	+ $\sum_{mi \in M(ca)} \lvert Par(m_i) \rvert$

Tableau 33. Impact de la conversion d'une relation sur les métriques de sous-classe

Métrique	Variation de valeur
NMA	+ [2 × NAI]
NAM	+ [2 × NAI]
DAM	− [NAI/NTA]
OAM	− [2×OAM× NAI/(NMA + 2×NAI)]
NPA	− NAI
NOC	− 1

Tableau 34. Impact de la conversion d'une relation sur les métriques de super-classe initiale

Métrique	Variation de valeur
OCAEC	+1
OCMEC	+1

Tableau 35. Impact de la conversion d'une relation sur les métriques Ancêtre de super-classe initiale

Métrique	Variation de valeur
NOC	+ 1
OCAEC	− 1
OCMEC	− $\sum_{mi \in M(ca)} \lvert Par(m_i) \rvert$

Tableau 36. Impact de la conversion d'une relation sur les métriques de nouvelle super-classe

Métrique	Variation de valeur
OCAEC	– 1
OCMEC	– 1

Tableau 37. Impact de la conversion d'une relation sur les métriques de l'ancêtre de nouvelle super-classe

3.4 CONCLUSION

Dans les sections précédentes, nous avons commencé par présenter une liste partielle des métriques de stabilité, utilisées dans la partie expérimentation, ainsi que leurs définitions. Ensuite, nous avons abordé les deux formes de restructurations avec leur définition. Des relations de cause à effet ont été établies entre une transformation de bas ou de haut niveau et les valeurs de métriques. Ces relations ont un rôle crucial lors de la suggestion des alternatives de design, car elles guident le processus d'amélioration de la qualité.

Le chapitre suivant est consacré à l'approche sur laquelle l'outil OO1_correct se base.

Chapitre 4

UNE APPROCHE POUR LA RESTRUCTURATION AUTOMATIQUE DE LOGICIELS OO

Les travaux précédents sur l'amélioration de la qualité des systèmes à objets proposent, d'une part, des solutions permettant la détection des anomalies de conception à l'aide des modèles prédictifs, et d'autre part, des restructurations élémentaires ou complexes permettant d'améliorer la qualité. Cependant, ces deux aspects ont été traités, dans la plupart des études, indépendamment l'un de l'autre.

L'idée derrière la technique présentée dans ce chapitre est de combiner ces deux familles de travaux afin d'obtenir une seule approche. En effet, l'approche proposée permet la détection automatique des anomalies de conception, où une restructuration particulière pourra être appliquée pour améliorer la qualité. Ce processus est basé sur la mesure de l'impact de restructurations sur les valeurs de métriques logicielles, telle qu'elle est présentée dans le chapitre précédent.

Ce chapitre décrit donc en détails la démarche reliée à notre approche.

4.1 RÉSUMÉ DE L'APPROCHE

La figure 13 illustre les mécanismes de prédiction et suggestion.

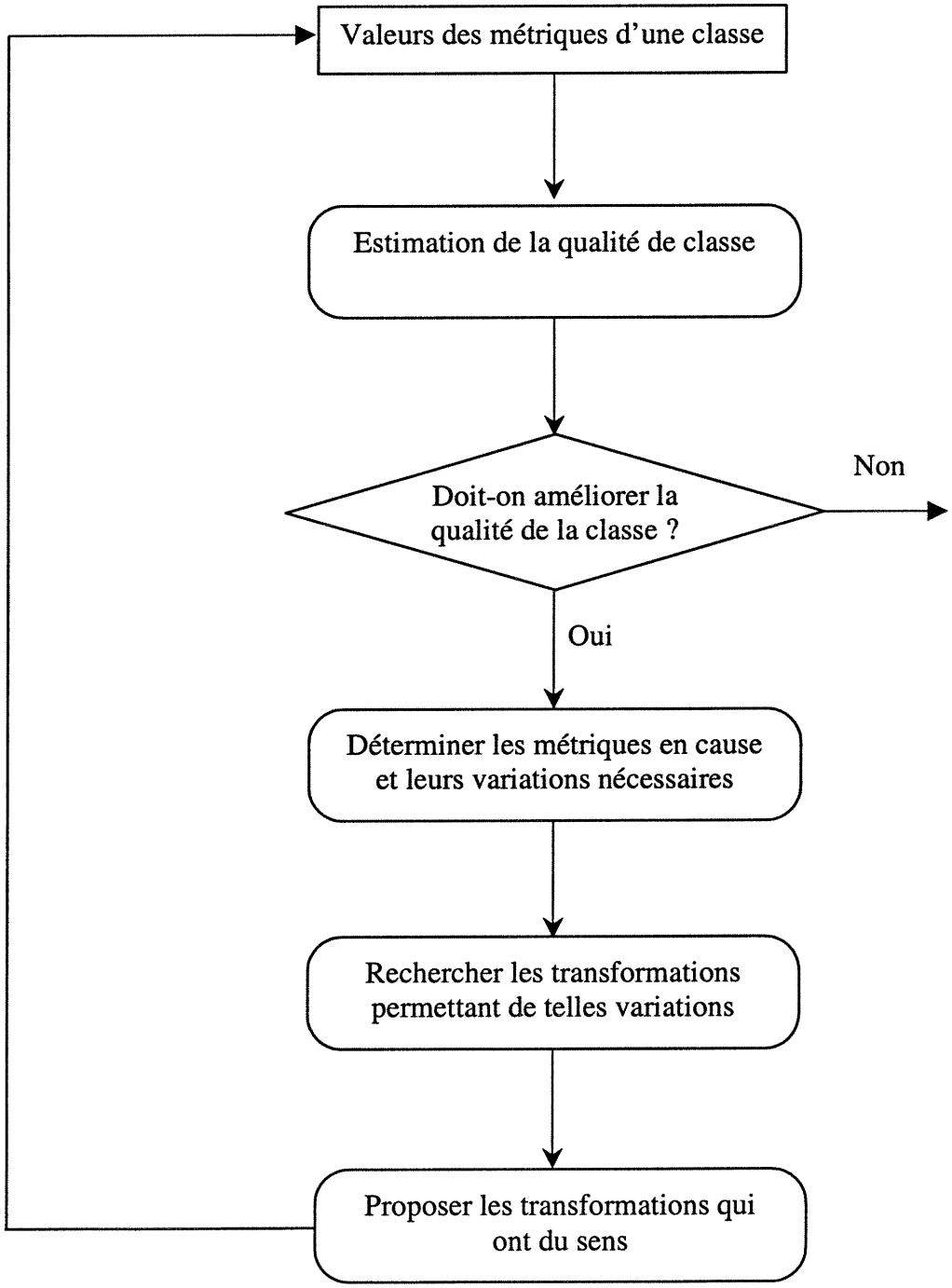


Figure 13. Vue d'ensemble de l'approche

Dans ce travail, nous nous sommes intéressés à la prédiction et à l'amélioration de la stabilité de classes, au moyen de modèles prédictifs flous. Cependant, cette technique reste valide pour tout autre critère de la qualité (maintenabilité, réutilisabilité, fiabilité,...). Elle s'applique également aux modèles prédictifs en général, y compris les réseaux bayésiens, les modèles de régression linéaires, etc.

Notre technique est analogue à celle d'un correcteur grammatical ou stylistique d'un logiciel de traitement de texte. Après avoir détecté les anomalies dans le texte écrit, le correcteur de style suggère à l'utilisateur des changements, en se basant sur des règles de grammaire ou de style bien défini. L'utilisateur a tout le choix d'accepter ou de rejeter les suggestions.

Dans notre cas, ce processus implique quatre étapes :

i) Estimation de la qualité à l'aide des modèles de prédiction floue : ceci permettant d'obtenir la qualité (bonne/mauvaise qualité, degré de vérité) de chaque classe de l'application en question ;

ii) Sélection des métriques qui ont causé les anomalies détectées et des possibilités d'ajustements qu'offrent leurs fonctions d'appartenance ;

iii) Identification des restructurations susceptibles d'apporter aux valeurs des métriques les modifications nécessaires telles que spécifiées dans ii) ;

iv) Ajustement ou raffinement du choix de restructurations sélectionnées auparavant, en les confrontant conceptuellement et structurellement au système étudié ; nous retenons par la suite celles qui seront applicables au contexte de l'application.

Les sections suivantes décrivent en détail les étapes de l'approche.

4.2 DESCRIPTION DÉTAILLÉE DE L'APPROCHE

4.2.1 Estimation de la qualité

4.2.1.1 Fuzzyfication de métriques

Pour qu'on puisse appliquer les règles d'un modèle de prédiction flou à une classe donnée, nous devons tout d'abord fuzzyfier les valeurs concrètes des métriques de la classe en question. L'opération de fuzzyfication permet de transformer les valeurs concrètes des métriques en des étiquettes avec degrés de vérité, par l'évaluation des fonctions d'appartenance[11] définies sur ces métriques. Cette opération donne naissance à trois couples ayant la forme *(NomMétrique étiquettes* [12] *; DegréDeVérité)*, qui sont associés à chaque valeur concrète fuzzyfiée. On dit que cette démarche détermine le degré d'appartenance d'une valeur concrète à un ensemble flou.

Exemple (cf. figure 14) : la fuzzyfication de la valeur *DIT = 2* permet d'obtenir les trois couples suivants : *(DIT Small ; 0.7), (DIT Medium ; 0.4) et (DIT High ; 0).*

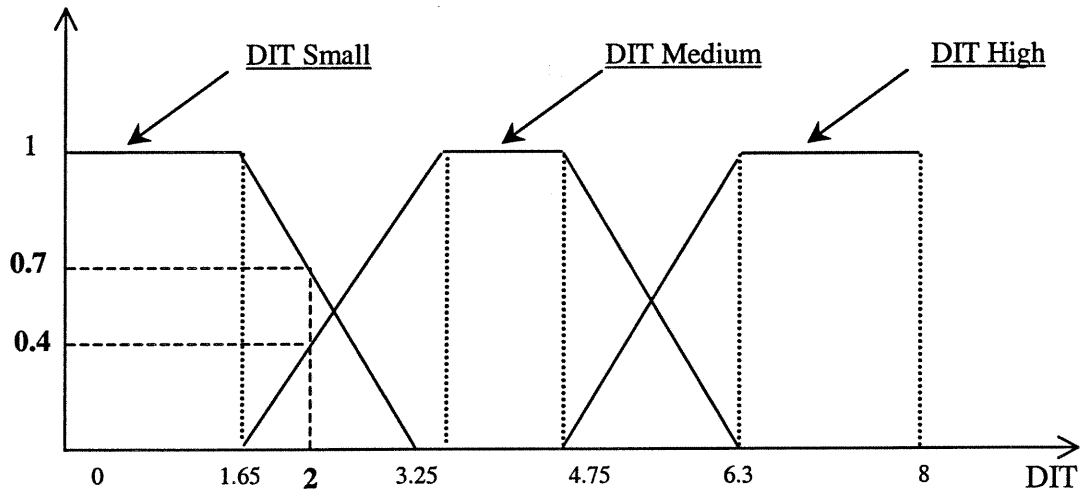


Figure 14. Exemple de fuzzyfication de la métrique DIT

[11] Les fonctions d'appartenance utilisées dans ce mémoire ont été définies dans [SAR01].
[12] Comme par exemple *"Small"*, *"Medium"*, *"High "* ou « Petit », « Moyen », « Grand », ...

On peut aussi dire que la proposition *DIT Small* est vraie à 70 %. On parle alors du degré de vérité de la proposition. Le degré d'appartenance et le degré de vérité sont donc des notions similaires.

4.2.1.2 Évaluation de règles

Une fois que l'opération de fuzzyfication est terminée, l'étape suivante consiste à appliquer les règles du modèle de qualité sur les classes du système. Le déclenchement des règles permet d'obtenir la qualité des classes du système[13].

Un modèle de prédiction flou est un ensemble de règles écrites selon le formalisme de la logique floue, qui établi une relation de cause à effet entre deux caractéristiques du logiciel : les attributs externes et les attributs internes. Un attribut externe est mesuré d'après la façon dont le produit interagit avec son environnement [FEN91]. La possibilité de mise à l'essai, la fiabilité, la stabilité et la maintenabilité sont des exemples d'attributs externes. Un bon nombre des attributs externes ne peuvent être mesuré directement, jusqu'à très tard au cours d'un projet ou même au cours du cycle de vie d'un produit. Les attributs internes, tels que la profondeur de l'arbre d'héritage (*"Depth in Inheritance Tree"ou DIT*) ou encore le nombre d'enfants d'une classe (*"Numbre Of Children" ou NOC*), sont directement mesurables, et ils sont utilisés comme indicateurs avancés des attributs externes.

Une règle prédictive floue, R_f, est une expression de la forme :

R_f: *IF <Fuzzy proposition> THEN <conclusion>.*

La proposition floue (ou prémisse) permet de représenter en termes logiques l'état d'une variable linguistique. Cette proposition peut représenter une condition (dans ce cas, la variable linguistique est une métrique) ou une conclusion intermédiaire utilisée dans la partie gauche d'une règle comme condition (dans ce cas, la variable linguistique est le nom d'un critère intermédiaire de la qualité). Les versions atomique et composée

[13] Nous avons opté pour l'utilisation des opérateurs de Zadeh [ZAD65] : Le t-norme minimum (ET-*and*) est utilisé entre les conditions d'une règle, et le s-norme maximum (OU-*or*) est utilisé dans le cas où plusieurs règles concluent à la même conclusion.

constituent les deux types de propositions floues usuelles. La première, comme son nom l'indique, se compose d'un seul terme et s'exprime simplement de la façon suivante :

M is A,

Où M représente le nom d'une métrique (appelée aussi variable linguistique) et A, la valeur ou l'état de cette métrique (appelé aussi attribut qualitatif).

Exemple : la proposition *"DIT is Small"* stipule que la métrique *DIT* se trouve dans l'état *"Small"* et se représente par la fonction d'appartenance $\mu_{small}(DIT)$.

La proposition composée, quant à elle, se veut être un amalgame de diverses propositions floues assemblées par le connecteur logique du type t-norme ET. Par exemple, la proposition composée suivante unit les deux métriques *DIT* et *NMA* de valeurs qualitatives respectives *"Small"* et " *Medium*" par l'opérateur floue ET (*and*) :

(DIT is Small) and (NMA is Medium).

Une conclusion peut représenter l'évaluation intermédiaire ou finale de la qualité d'une classe. Elle s'exprime de la même manière qu'une condition, à la différence qu'il n'y a pas de fonctions d'appartenance définie sur une conclusion. Les deux règles suivantes donnent l'exemple d'une conclusion intermédiaire et finale :

R_I: *IF (OCAEC is LARGE) THEN (coupling_impact is LARGE);*

R_F: *IF (negative_change_effect is LARGE) THEN (Stability is SMALL);*

4.2.2 Sélection des métriques à l'origine de mauvaise qualité et leurs variations nécessaires

Cette étape correspond à la sélection de métriques à l'origine de mauvaise qualité. De ce fait, pour chaque règle dont la conclusion aboutit à une conclusion finale, nous explorons les conditions de cette règle en comparant leurs valeurs avec celle de la conclusion finale, et nous retenons les prémisses ayant la même valeur de vérité que celle de la conclusion finale. Ensuite, de chaque prémisse retenue, nous retirons la variable linguistique (c'est-à-

dire le nom d'une métrique); nous obtenons ainsi l'ensemble des métriques qui ont causé la mauvaise qualité.

Selon les formes de fonctions d'appartenance définies sur les métriques, nous pourrions déterminer si les valeurs de ces métriques devaient être augmentées ou diminuées, et dans quelles limites, afin d'améliorer le degré de vérité d'une conclusion finale.

Nous discuterons, dans ce qui suit, des possibilités de changements offertes par les quatre formes de fonction d'appartenance. Les formes sont : triangulaire, trapézoïdale, trapézoïdale à droite ("*right trapezoidal*") et trapézoïdale à gauche ("*left trapezoidal*").

Supposons que la prémisse P = M *is* A est détectée comment ayant un degré de vérité identique à celui d'une conclusion donnée. M désigne ainsi la métrique à l'origine de mauvaise qualité et v_{CM} est sa valeur concrète.

4.2.2.1 Les variations données par la forme triangulaire

Une fonction d'appartenance triangulaire (cf. figure 15) est définie par les trois points x_1, x_2 et x_3. Ces points sont appelés par la suite points d'inflexion.

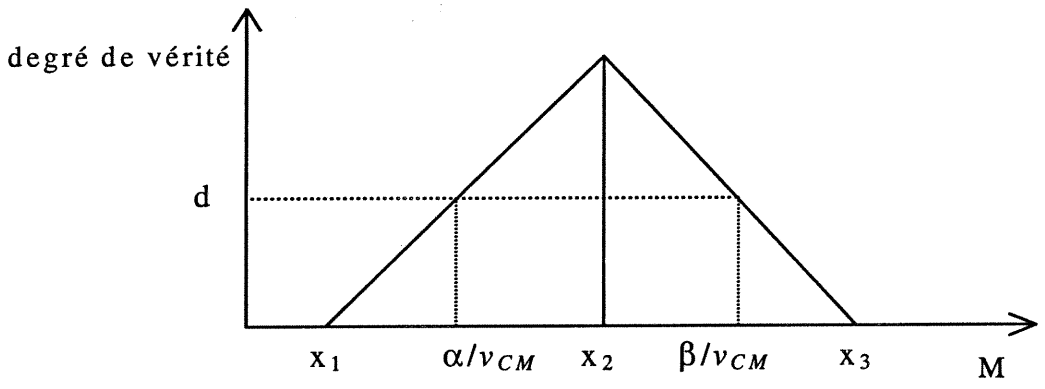


Figure 15. La forme triangulaire d'une fonction d'appartenance

Nous distinguons deux cas dans lesquels la valeur concrète v_{CM} peut être présente :

i- $x_1 \leq v_{CM} \leq x_2$:

Dans cet intervalle, augmenter le degré de vérité de P revient à augmenter la valeur concrète v_{CM} de la métrique M. À cette fin, nous devrons trouver au moins une

restructuration capable d'apporter une augmentation à v_{CM}. Après l'application d'une telle restructuration, v_{CM} doit se trouver dans l'intervalle $[v_{CM},\beta]$. Ici, β représente le point situé entre x_2 et x_3, ayant la même valeur de vérité que celle de v_{CM}.
La diminution du degré de vérité de P, quant à elle, sera possible soit en diminuant v_{CM} (on l'approchera le plus possible de x_1 au moyen d'une restructuration convenable); soit en augmentant v_{CM} (on la remettra dans l'intervalle [β, x3]). On désigne par β le point situé dans l'intervalle $[x_2, x_3]$ et ayant le même degré de vérité que v_{CM}.

ii- $x_2 \leq v_{CM} \leq x_3$:

Pour augmenter le degré de vérité de P, nous devrons diminuer la valeur concrète v_{CM} de la métrique en question. Une restructuration susceptible d'apporter une diminution à v_{CM} doit être en mesure de la faire entrer dans l'intervalle $[\alpha, v_{CM}]$; où α représente le point situé dans l'intervalle $[x_1, x_2]$, ayant la même valeur de vérité que celle de v_{CM}.

Pour ce qui est de la diminution du degré de vérité de P, elle sera possible dans deux cas. Le premier est l'augmentation est l'augmentation de v_{CM}, de telle manière qu'elle rapproche le plus possible du point d'inflexion x_3, au moyen d'une restructuration convenable. La deuxième possibilité de diminuer la valeur de P est de diminuer v_{CM} à l'aide d'une restructuration appropriée, de telle sorte que la nouvelle valeur de v_{CM} sera trouvée dans l'intervalle $[x_1,\alpha]$; α désigne le point situé entre les deux points d'inflexion x_1 et x_2 et ayant le même degré de vérité que v_{CM}.

4.2.2.2 Les variations données par la forme trapézoïdale

Une fonction d'appartenance trapézoïdale (cf. figure 16) est définie par les quatre points d'inflexion x_1, x_2, x_3 et x_4.

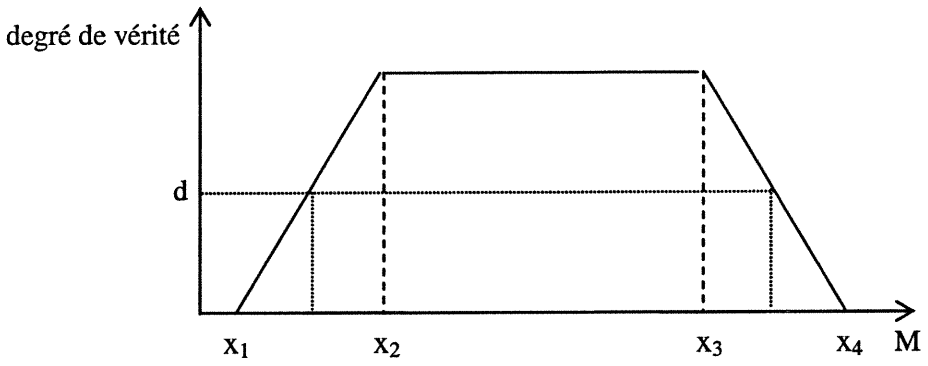


Figure 16. La forme trapézoïdale d'une fonction d'appartenance

Nous distinguons trois cas dans lesquels la valeur concrète v_{CM} peut être présente :

i- $x_1 \leq v_{CM} \leq x_2$:

C'est le même raisonnement que celui d'une forme triangulaire où v_{CM} appartient à l'intervalle $[x_1, x_2]$.

ii- $x_2 \leq v_{CM} \leq x_3$:

Dans cet intervalle, la valeur de vérité de P est déjà maximale (égale à 1). La seule possibilité de changement se résume ainsi à la diminution de cette valeur. Cette diminution sera possible soit en diminuant v_{CM}, en amenant sa valeur dans l'intervalle $[x_1, x_2]$, soit en augmentant v_{CM} en l'entrant dans l'intervalle $[x_3, x_4]$, au moyen d'une restructuration convenable.

iii- $x_3 \leq v_{CM} \leq x_4$:

C'est le même raisonnement que celui d'une forme triangulaire et où v_{CM} est appartient à l'intervalle $[x_2, x_3]$.

4.2.2.3 Les variations données par la forme trapézoïdale à droite

Une fonction d'appartenance trapézoïdale à droite, ou "*right trapezoidal*", est définie par les trois points d'inflexion x_1, x_2, et x_3, comme l'illustre la figure 17 ci-dessous.

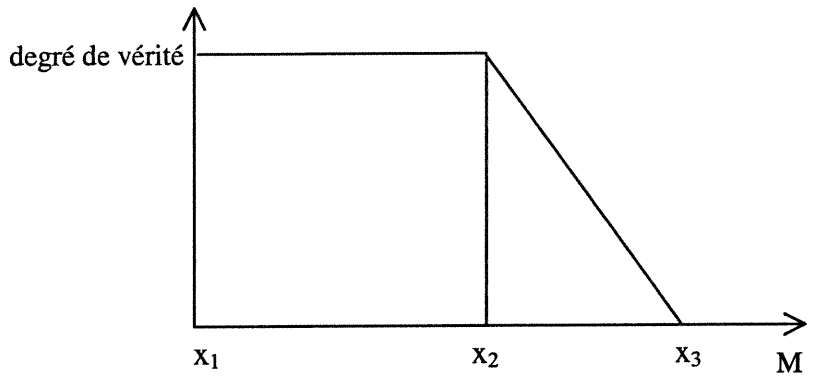


Figure 17. La forme trapézoïdale à droite d'une fonction d'appartenance

Nous considérons deux intervalles dans lesquels la valeur concrète v_{CM} peut exister :

i- $x_1 \leq v_{CM} \leq x_2$:

Dans cet intervalle, nous remarquons que la possibilité de changement de la valeur de vérité de *P* est restreinte à sa diminution, étant donné que ce degré de vérité est déjà maximal (égale à 1). Pour ce faire, nous devrons trouver au moins une restructuration susceptible d'augmenter la valeur v_{CM} de métrique et de la faire entrer dans l'intervalle $[x_2, x_3]$.

ii- $x_2 \leq v_{CM} \leq x_3$:

Pour augmenter le degré de vérité de *P*, nous devrons diminuer la valeur concrète v_{CM} de métrique, au moyen d'une restructuration convenable. Une telle restructuration devra apporter une diminution à v_{CM} de manière à entrer dans l'intervalle $[x_1, x_2]$.
Quant à la diminution du degré de vérité de *P*, elle sera remarquable dès que la valeur concrète v_{CM} de la métrique augmente. Le choix d'une restructuration susceptible d'apporter une telle augmentation à v_{CM} est guidé par sa capacité à approcher v_{CM} le plus possible du point d'inflexion x_3.

4.2.2.4 Les variations données par la forme trapézoïdale à gauche

Une fonction d'appartenance trapézoïdale à gauche (cf. figure 18), ou "*left trapezoidal*", est définie par les trois points d'inflexion x_1, x_2 et x_3.

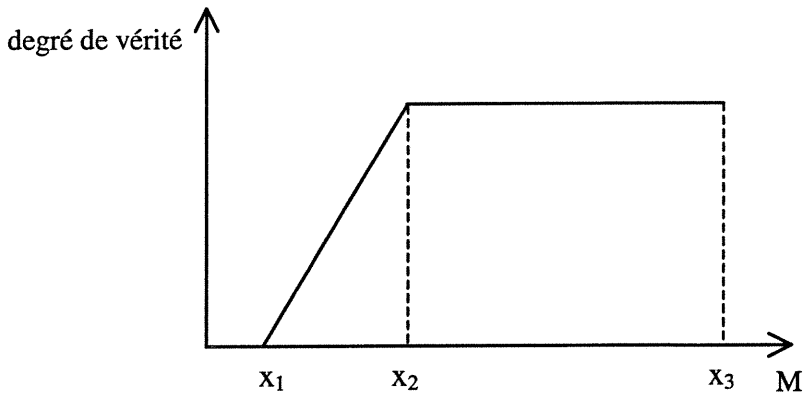


Figure 18. La forme trapézoïdale à gauche d'une fonction d'appartenance

La valeur concrète v_{CM} pourra être présente dans deux intervalles comme suit :

i- $x_1 \leq v_{CM} \leq x_2$:

L'augmentation du degré de vérité de P aura lieu si la valeur concrète v_{CM} de métrique est augmentée. De ce fait, nous devrons trouver au moins une transformation qui sera capable d'augmenter v_{CM}, en la rentrant dans l'intervalle $[x_2, x_3]$.
La diminution du degré de vérité de P exige la diminution de valeur concrète v_{CM} de métrique. Une restructuration susceptible d'apporter une diminution à v_{CM} doit être en mesure de la faire entrer dans l'intervalle $[x_2, x_3]$.

ii- $x_2 \leq v_{CM} \leq x_3$:

Dans cet intervalle, la valeur de vérité de P est déjà maximale (égale à 1). La possibilité de changement est ainsi restreinte à la diminution de cette valeur. Pour ce faire, nous devrons trouver au moins une restructuration susceptible d'apporter une diminution à la valeur v_{CM} de métrique, faisant entrer cette valeur dans l'intervalle $[x_1, x_2]$.

Nous pouvons récapituler tous les cas étudiés précédemment dans le tableau ci-dessous. *Aug.* (v_{CM}) et *Dim.* (v_{CM}) désignent respectivement l'augmentation et la diminution de valeur concrète v_{CM} de la métrique M.

	Forme de fonction	Valeur concrète v_{CM}		
		$x_1 - x_2$	$x_2 - x_3$	$x_3 - x_4$
Augmentation du degré de vérité d'une prémisse P	Triangulaire	*Aug.* (v_{CM})	*Dim.* (v_{CM})	-
	Trapézoïdale	*Aug.* (v_{CM})	-	*Dim.* (v_{CM})
	Trap. à droite	-	*Dim.* (v_{CM})	-
	Trap. à gauche	*Aug.* (v_{CM})	-	-
Diminution du degré de vérité d'une prémisse P	Triangulaire	*Aug.* (v_{CM}) ou *Dim.* (v_{CM})	*Aug.* (v_{CM})	-
	Trapézoïdale	*Dim.* (v_{CM})	*Aug.* (v_{CM}) ou *Dim.* (v_{CM})	*Aug.* (v_{CM})
	Trap. à droite	*Aug.* (v_{CM})	*Aug.* (v_{CM})	-
	Trap. à gauche	*Dim.* (v_{CM})	*Dim.* (v_{CM})	-

Tableau 38. Tableau récapitulatif des possibilités de changement selon les formes de fonction d'appartenance définies sur les métriques

4.2.3 Identification de restructurations appropriées

Après avoir détecté les situations symptomatiques en utilisant les règles d'un modèle de qualité et que les métriques à l'origine de ces situations aient été localisées, l'étape suivante sera d'identifier des restructurations susceptibles d'améliorer la qualité, tout en conservant le comportement du programme. Le principe de suggestion d'une restructuration se base sur l'établissement d'une relation entre les transformations et l'amélioration de la qualité. Pour dériver une telle relation, nous utilisons l'hypothèse suivante : si une bonne conception correspond à une bonne combinaison des valeurs des

métriques, alors il y a de fortes chances qu'une bonne combinaison de valeurs des métriques corresponde à une bonne conception du système [SAH00].
Le processus d'identification est fait en comparant la direction (augmentation ou diminution) et l'amplitude (intervalle) des variations des métriques, sélectionnées auparavant, avec les variations fournies par les quatre restructurations de haut niveau, étudiées au chapitre 3. Celles qui satisfont aux changements exigés sont retenues en tant que transformations potentielles.

Exemple : pour augmenter la valeur concrète de métrique *DIT*, sélectionnée comme étant la cause d'une mauvaise qualité d'une classe *c*, deux restructurations ont été trouvées. La première consiste à la création d'une classe abstraite d'un ensemble de classes cibles. Au sein de cette restructuration, la classe *c* doit jouer le rôle des classes cibles pour que la valeur de *DIT* augmente d'un point (DIT -> *DIT +1)*. La deuxième restructuration consiste à la conversion de relation d'héritage en une agrégation. Pour que cette transformation puisse augmenter la valeur de *DIT*, la classe *c* doit jouer le rôle de sous-classe dans la relation à convertir. Ainsi, ces deus restructurations seront retenues comme des restructurations potentielles.

4.2.4 Sélection de restructurations applicables au contexte

Cette étape consiste à raffiner les restructurations identifiées dans l'étape précédente, en gardant celles qui sont applicables au contexte du système en question. Pour cette raison, les suggestions résultantes devront être confrontées conceptuellement et structurellement au système étudié, afin de déterminer celles qui peuvent être retenues et pouvoir les expliciter avec précision. Par exemple, pour qu'on puisse factoriser une classe, il faut trouver d'autres classes du système ayant des comportements en commun. Puis l'on spécifiera les méthodes à abstraire ou à créer dans la nouvelle super-classe. Ou bien, si la restructuration était une conversion de la relation d'héritage en une relation d'agrégation, alors il faut que la classe en question ait déjà une super-classe.

La réalisation de cette étape peut impliquer une intervention humaine ou peut être parfois automatisée jusqu'à un certain point.

4.3 CONCLUSION

Ce chapitre a décrit notre approche à la restructuration automatique de logiciels orientés objet. Tel que présenté, l'utilisation des modèles de prédiction flous a pour effet d'augmenter les choix de modifications de valeur des métriques. Ceci augmente la probabilité de trouver des restructurations convenables et qui seront capable de changer la valeur de métriques et, par la suite, d'améliorer les qualités des classes.

Le chapitre suivant porte sur la mise en œuvre de cette approche à travers le prototype OO1_Correct.

Chapitre 5

LA MISE EN ŒUVRE DE L'APPROCHE AU MOYEN DE OO1_CORRECT

Pour être en mesure de supporter l'approche précédemment décrite, un outil qui automatise les tâches de détection des anomalies et de suggestion des alternatives de conception est nécessaire.

Le présent chapitre porte sur la partie floue de l'outil « OO1_Correct[14] », développé afin de supporter notre technique. Il explore l'architecture logicielle de l'outil en mettant l'accent, entre autres, sur ses caractéristiques et les modules dont il est composé. Nous décrivons ensuite l'environnement de réalisation et d'implantation (langage de modélisation et d'implantation, outils de développement) ainsi que les motivations de ces choix.

[14] OO1_correct pour Object Oriented Correct, l'outil englobe l'approche classique et floue de prédiction et de restructuration. Dans ce document, nous restreindrons notre description à la partie basée sur la logique floue.

5.1 CARACTERISTIQUES DE OO1_CORRECT

OO1_Correct est un outil qui peut être utilisé aussi bien pour estimer la qualité que pour suggérer des alternatives de conception, selon les approches classique et floue.
Ses principales caractéristiques (ou fonctionnalités) sont :

i) Facilité d'utilisation et capacité de gérer des nouvelles données d'entrée (modèles prédictifs, systèmes à évaluer, etc.) après le lancement de l'outil, par l'intermédiaire de son interface graphique. Il n'est donc pas nécessaire de relancer l'outil pour chaque nouveau modèle de qualité ou pour chaque système.

ii) La possibilité d'évaluer l'efficacité de l'approche sur laquelle l'outil se base pour prédire la qualité (l'option « évaluation » dans l'interface de l'outil).

iii) La mise à disposition des traces d'exécution : l'outil offre la possibilité d'obtenir des rapports en format XML/HTML décrivant les résultats d'évaluations et de suggestions. Ces rapports seront très utiles pour une analyse ultérieure des résultats obtenus.

iv) Intégration de base de données : l'outil offre la possibilité de stocker les données d'entrée de traitement (modèle de qualité, valeurs de métriques, fonctions d'appartenance, etc.) dans une base de données relationnelle. Ceci permet d'assurer une organisation efficace des données, en réduisant la redondance et en augmentant le temps de traitement et de mise à jour.

À l'heure actuelle, l'outil tourne sur la plate-forme *Windows*, y compris *Win* 9.x, ME, 2000 et *NT* 4. La portabilité de Java permet d'exécuter cette application sur des machines utilisant *Solaris* et *Linux*. Cependant, l'étude de cas présentée dans la deuxième partie du chapitre actuel est effectuée dans l'environnement *Windows* 2000.

5.2 ARCHITECTURE DE L'OUTIL

Celle-ci consiste en cinq modules interactifs (cf. figure 19) : Fuzzyfication, Parser, Moteur d'inférence, Estimation de la qualité et le module Restructuration.

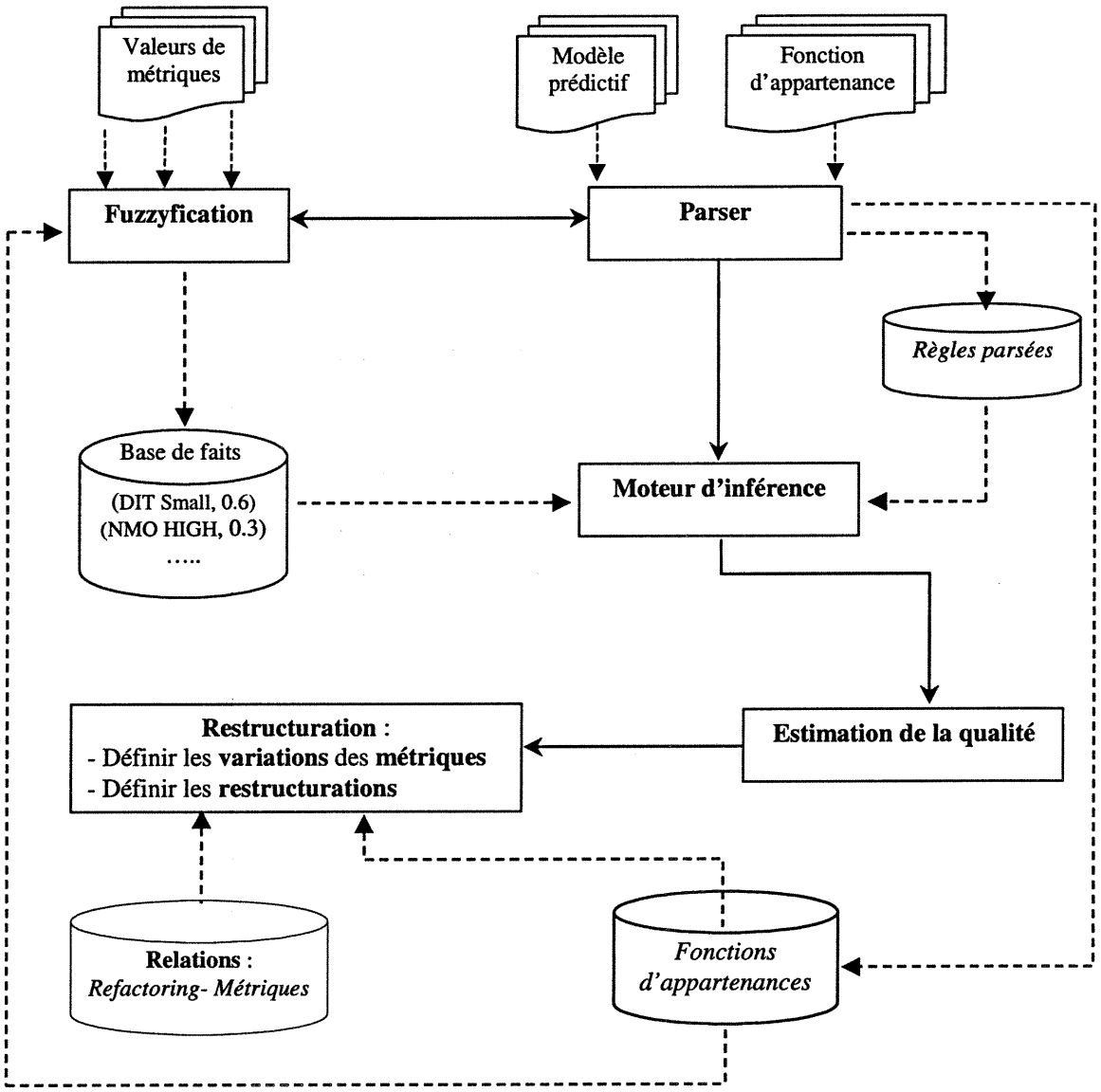


Figure 19. L'architecture générale de OO_Correct

Le traitement commence par la fuzzyfication des valeurs concrètes de métriques des classes à améliorer leur qualité. Cette opération permet de créer la base de faits et

l'enrichir au fur et à mesure que la fuzzyfication se progresse. Ensuite, le parser prend la relève, en transformant les règles du modèle prédictif ainsi que les fonctions d'appartenance définis sur les métriques, en un format interne, facile à gérer par le moteur d'inférence. Les résultats de parsing seront stockes dans une base de données, accessible par le moteur d'inférence. Une fois le travail du parser est terminé, le moteur d'inférence exécute les règles du modèle, en permettant ainsi l'évaluation de la qualité des classes en question. L'ultime étape consiste à restructurer les classes jugées comment ayant une mauvaise qualité. Ces modules constituent notre majeure contribution à la construction de l'outil OO1_Correct.

5.3 DESCRIPTION DES DIFFÉRENTS MODULES

5.3.1 Module Fuzzyfication

Ce module représente l'ensemble de classes qui traitent du passage du domaine réel ou concret au domaine flou. En effet, l'opération de fuzzyfication consiste à transformer la valeur numérique d'une métrique d'entrée en un degré d'appartenance flou par l'évaluation de la fonction d'appartenance[15], définie sur cette métrique. Ceci résulte en trois couples de la forme "*nom_de_métrique étiquette*; *degré_de_vérité*". Ces couples seront ajoutés à la base de faits, créée au début de traitement, et seront utilisés par le moteur d'inférence lors du déclenchement des règles. Par exemple (cf. figure 20), dans le cas où la valeur courante de la métrique *DIT* serait de 2, le degré d'appartenance à la fonction d'appartenance "*DIT Small*" est égal à 0.7; celui de la fonction d'appartenance "*DIT Medium*" est égal à 0.3; et celui de la fonction d'appartenance "*DIT High*" est égal à 0.0. La base des faits contient ainsi trois éléments : (*DIT Small; 0.7*), (*DIT Medium; 0.3*) et (*DIT High; 0.0*).

[15] Voir [SAR01] pour les détails concernant la définition du vocabulaire et des formes des fonctions d'appartenance.

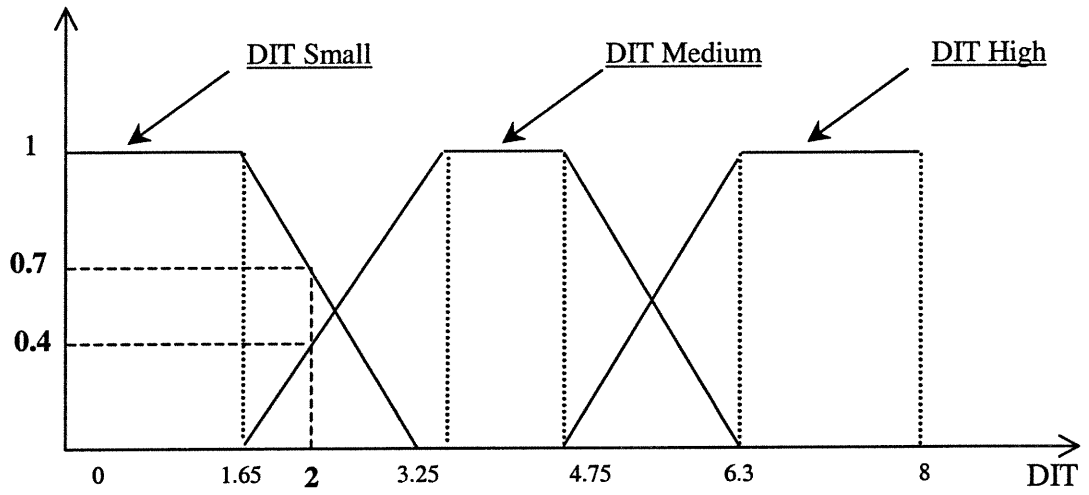


Figure 20. Fuzzyfication de valeurs concrètes

5.3.2 Module Parser

Ce module représente l'analyseur syntaxique, ou parser tout court, des fonctions d'appartenance et des règles de prédiction. Le parser traduit ces données en une forme exploitable par l'outil, en générant les objets y correspondant.

❖ Parsing d'une fonction d'appartenance

Une fonction d'appartenance, associée à une métrique *M*, se décrit sous la forme suivante :

Metric_Name;

$(MSF\_Form)_i$; $(Label)_i$; $(InfPt_j)_i$; $(\mu_M(InfPt_j))_i$,

Où :

Metric_Name : désigne le nom de la métrique,

MSF_Form : représente la forme d'une fonction d'appartenance,

Label : désigne l'étiquette associée à MSF_Form,

$InfPt_j$: les points d'inflexion de MSF_Form,

$\mu(InfPt_j)$: le degré d'appartenance à MSF_Form du point d'inflexion $InfPt_j$,

$j = 1, \ldots 4, i = 1, \ldots, 3$

L'analyse d'une telle fonction d'appartenance se déroulera comme suit : à l'aperçu du fragment *Metric_Name*, le parser génère deux objets, l'un de type *Input_Metric*, l'autre de type *MemberShipFunction*. La création de ce dernier objet donne naissance à trois autres objets, tous de même type, soit *FunctionForm*. Ces objets seront spécialisés et ensuite étiquetés, dès que le parser détecte respectivement le fragment du texte *MSF_Form* et *Label*. Les objets représentant les points d'inflexion d'un objet *FunctionForm*, ainsi que les degrés d'appartenance qui y sont associés, seront créés respectivement à la détection du fragment du texte $InfPt_j$ et $(\mu_M (InfPt_j))$.

❖ Parsing d'une règle floue :

Une règle de prédiction floue s'exprime de la manière suivante :

RuleName: *IF <M is A> THEN <C is B>*

Le parsing d'une règle R_f s'amorce par la création d'un objet de type *Rule* à la détection du nom de cette règle. Puis, à l'aperçu du mot clé *IF* (et ultérieurement *THEN*), un objet de type *FuzzyExpression* prend naissance. L'objet en découlant fera l'état d'une spécialisation ultérieure et sera transformé en un objet de type *condition* ou *conclusion*, selon qu'il représente une condition, une conclusion intermédiaire ou encore une conclusion finale. Ensuite, deux objets de types respectifs *LinguisticVariable* et *LinguisticTerm* seront créés à la détection des expressions *M is A* (et ultérieurement *C is B*).

5.3.3 Module Moteur d'inférence

Le moteur d'inférence s'occupe de la déduction logique de règles de prédiction. Il permet d'évaluer la qualité des systèmes, par l'entremise des valeurs fuzzyfiées des métriques et des règles de prédictions déjà parsées. La figure 21 ci-dessous résume le fonctionnement du moteur d'inférence. Dans le cas où il y aurait plusieurs règles qui se concluent sur une même conclusion, on commence avec la première qu'on en a trouvé. Ici, on suppose que la conclusion d'une règle et sa négation n'existe pas dans le modèle de prédiction.

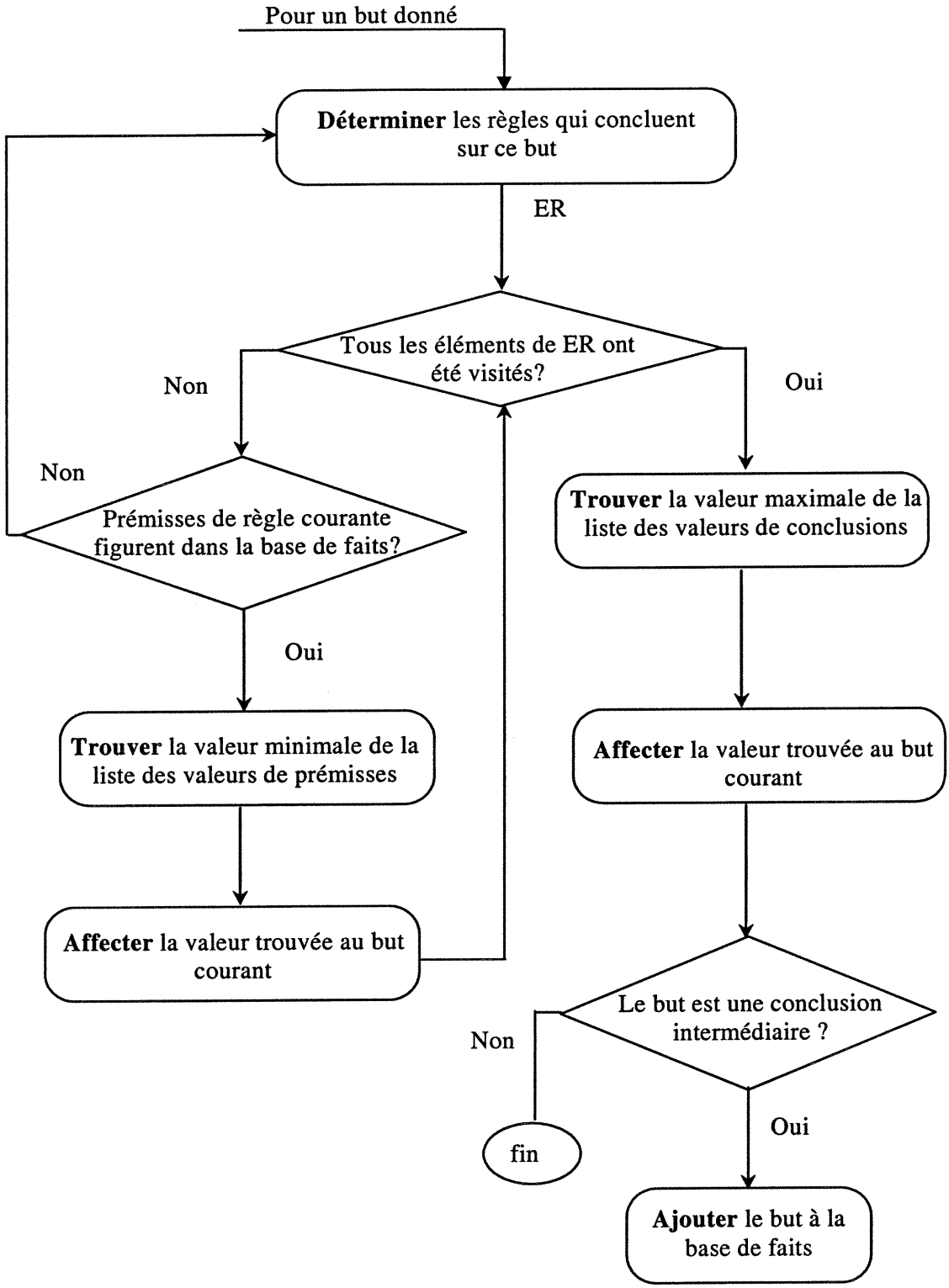


Figure 21. Fonctionnement du moteur d'inférence

La stratégie de raisonnement adoptée par le moteur d'inférence est celle du chaînage mixte[16], où le chaînage arrière fait appel au chaînage avant pour déclencher une règle. Le processus d'inférence s'amorce par la sélection des règles aboutissant à une conclusion finale. Ensuite, le moteur tente d'exécuter chacune des règles en vérifiant si toutes leurs prémisses figurent dans la base de faits. À cet effet, nous considérons les deux cas suivants :

i- Toutes les prémisses de la règle figurent dans la base de faits : ceci conduit au déclenchement des règles en question et par la suite, à l'affectation de valeurs minimales des prémisses à sa conclusion[17].

ii- Parmi les prémisses, il y en a une ou plusieurs qui ne figurent pas dans la base de faits : dans ce cas, la prémisse n'est autre qu'une conclusion intermédiaire utilisée comme condition dans la partie gauche de la règle. Le moteur considère cette conclusion comme un nouveau but et essaie de calculer son degré de vérité, en procédant à la recherche des règles qui conduisent à ce but, comme s'il s'agissait d'une conclusion finale.

Une fois que les règles concluant sur un but donné ont été déclenchées, le moteur d'inférence lui affecte la valeur maximale parmi les valeurs de conclusions de règles. S'il s'agissait d'une conclusion intermédiaire, elle sera ajoutée à la base de faits.

Nous illustrons le raisonnement du moteur d'inférence au moyen de l'exemple suivant.

Exemple : pour faciliter la vie, nous considérons une base de règles constituées seulement de deux règles R_1 et R_2. IF *DAM is SMALL* THEN *potential_coupling is LARGE*

R_1: IF P_1 AND P_2 THEN *middleman _conclusion*
R_2: IF *middleman _conclusion* AND P_3 THEN *classification*

---

[16] On fait appel aux stratégies de raisonnement classiques puisque les conclusions n'ont pas de fonctions d'appartenance, et par conséquent les techniques d'un moteur d'inférence flou ne pourront être applicables.
[17] Le t-norme minimum (ET-*and*) est utilisé entre conditions, et le s-norme maximum (OU-*or*) est utilisé entre conclusions.

Au début, la base de faits contient les trois prémisses P_1, P_2 et P_3 et le but initial est la conclusion de règle R_2, *classification*. Soit v_1, v_2 *et* v_3 les valeurs respectives de prémisses P_1, P_2 et P_3 où : $v_1 > v_2 > v_3$;

- Base de faits = $\{P_1, P_2, P_3\}$;
- But donné = *classification* ;

ER (*but*) = $\{R_i /$ la conclusion de R_i est *but*$\}$

- Déterminer les règles qui concluent sur *classification* $\Rightarrow$ *ER* = $\{R_2\}$;
- Tous les éléments de *ER* ont été visités ? Non,
  - Prémisses de règle courante (qui est R_2) figurent dans la base de faits? Non, la prémisse *middleman _conclusion* n'existe pas dans la base.
    - Déterminer les règles qui concluent sur *middleman _conclusion* $\Rightarrow$ *ER'* = $\{R_1\}$
    - Tous les éléments de *ER'* ont été visités ? Non,
      - Prémisses de règle courante (qui est R_1) figurent dans la base de faits? Oui,
      - Trouver la valeur minimale parmi les valeurs de prémisses $\Rightarrow v_2$;
      - Affecter la valeur trouvée au but courant $\Rightarrow$ valeur temporaire de (*middleman _conclusion*) = v_2 ;
    - Tous les éléments de *ER'* ont été visités ? Oui,
      - Trouver la valeur maximale parmi la liste des valeurs de conclusions $\Rightarrow$ il n'y a qu'une seule règle et par suite une seule conclusion, et la valeur maximale est égale ainsi à v_2 ;
      - Affecter la valeur trouvée au but courrant $\Rightarrow$ valeur permanente de (*middleman _conclusion*) = v_2 ;
      - Le but courant est une conclusion intermédiaire ? Oui,
        - Ajouter le but à la base de faits $\Rightarrow$ Base de faits = $\{P_1, P_2, P_3,$ *middleman _conclusion*$\}$;
  - Prémisses de règle courante (qui est R_2) figurent dans la base de faits? Oui,

        - Trouver la valeur minimale parmi les valeurs de prémisses $\Rightarrow v_3$;
        - Affecter la valeur trouvée au but courant $\Rightarrow$ valeur temporaire de (*classification*) = v_3 ;
- Tous les éléments de *ER* ont été visités ? Oui,
    - Trouver la valeur maximale de la liste des valeurs de conclusions $\Rightarrow$ il n'y a qu'une seule règle et par suite une seule conclusion, et la valeur maximale est égale ainsi à v_3 ;
    - Affecter la valeur trouvée au but courant $\Rightarrow$ valeur permanente de (*classification*) = v_3 ;
    - Le but courant est une conclusion intermédiaire ? Non,
        - Fin de traitement.

Le raisonnement du moteur permet d'obtenir la valeur de la conclusion finale *classification*. Ceci permettra d'évaluer la qualité de la classe sur laquelle les règles seront appliquées.

5.3.4 Module Estimation de la qualité

Le présent module s'intéresse à l'estimation de la qualité des classes, en examinant les valeurs des conclusions déjà données par le moteur d'inférence. Cet exercice a pour but de connaître la direction de changement (augmenter ou diminuer), en préparant les classes pour l'étape de restructuration subséquente. Nous distinguons deux cas selon lesquels le traitement peut être différent :

i- Cas d'une conclusion positive : une conclusion est dite positive si son attribut qualitatif est de genre *Grand (Large)*, *Élevé (High)*, etc. Si une telle conclusion a un degré de vérité égal à 1, ceci implique que la classe a certainement une bonne qualité. Il ne sera pas nécessaire de modifier cette valeur pour améliorer la qualité. Dans le cas où cette valeur est plus petite que 1, elle sera prise en considération pour être modifiée durant l'étape de restructuration.

Exemple : considérons la règle ci-dessous :

IF (*negative_change_effect is LOW*) THEN (*Stability is LARGE*);

Cette règle aboutit à une conclusion positive (*Stability is LARGE*). Si le degré de vérité de cette conclusion est égal à 1, la classe sur laquelle cette règle est appliquée sera tout à fait stable (100 % stable); ainsi, il ne sera pas nécessaire d'augmenter ce degré de vérité puisqu'il est déjà optimal. En revanche, si la valeur était plus petite que 1, une augmentation sera envisagée.

ii- Cas d'une conclusion négative : une conclusion est dite négative si son attribut qualitatif est de genre Petit (*Small*), Bas (*Low*), etc. Si une telle conclusion a un degré de vérité égal à 0, ceci implique que la classe a une tendance à avoir une mauvaise qualité avec une probabilité égale à 0. Ainsi, il ne serait pas nécessaire de modifier cette valeur pour améliorer la qualité. Par contre, si cette valeur était plus grande que 0, elle serait prise en considération pour être modifiée durant l'étape de restructuration.

Exemple : considérons la règle ci-dessous :

IF *(negative_change_effect is LARGE)* THEN *(Stability is LOW);*

Cette règle aboutit à une conclusion négative (*Stability is LOW*). Si le degré de vérité de cette conclusion est égal à 0, la classe sur laquelle cette règle s'applique a une tendance à être instable, avec une probabilité égale à 0. Ainsi, il ne sera pas nécessaire de réduire ce degré de vérité puisqu'il est déjà minimal. En revanche, si la valeur était plus petite que 1, une augmentation serait envisagée.

5.3.5 Module Restructuration

Ce module contient l'ensemble des classes dédiées à la suggestion de transformations susceptibles, une fois appliquée, d'améliorer la valeur des conclusions sélectionnées auparavant. La phase de restructuration se déroulera en trois étapes :

i- Identification des métriques à l'origine d'une mauvaise qualité : pour chaque conclusion, nous commençons tout d'abord par l'identification des prémisses dont les valeurs sont identiques à celles de la conclusion. Ceci consiste à faire une démarche arrière dans les règles jusqu'à ce que toutes les prémisses soient retrouvées. À partir des prémisses, on retient les noms des métriques. Il en résulte de ceci un ensemble de métriques, qui sont à l'origine d'une mauvaise qualité.

ii- Identification des changements nécessaires : tel qu'il est discuté au chapitre 4, à partir de formes de fonctions d'appartenance associées aux métriques, nous pouvons déterminer si les valeurs des métriques qui ont à l'origine de mauvaise qualité devraient être augmentées ou diminuées, ainsi que les limites dans lesquelles elles doivent l'être, afin d'obtenir le niveau envisageable de stabilité de la classe en question. Les fonctions d'appartenance définies sur les 6 métriques, utilisées dans le modèle prédictif, sont données dans l'annexe.

iii- Sélection des transformations : une confrontation entre les changements envisagés par l'étape précédente et ceux qui sont offerts par les transformations de haut niveau (voir Chapitre 3), permettra de sélectionner les restructurations qui répondent aux changements envisagés. Tout dépend du rôle de la classe. Une restructuration sera acceptée ou abandonnée si la classe de l'application peut jouer un rôle identique à celui d'une classe dans ladite restructuration.

5.3.6 Module Stockage des données du système

À l'architecture de base qui vient d'être décrite s'ajoutent deux modules dont le but est de rendre le traitement plus efficace et les analyses ultérieures des résultats, plus faciles.
Le module stockage de données a pour rôle de stocker les données d'entrées de l'outil. Les principales entités du système nécessitant la persistance dans une base de données sont :

- Valeurs de métriques de classes candidates ;
- Règles de modèle de prédiction ;
- Fonctions d'appartenance associées aux métriques ;
- Les relations métriques - restructurations ;
- Les traces d'évaluation et de restructurations.

5.3.7 Module Génération des rapports[18]

Ce module regroupe les classes se consacrant à la génération des rapports, en format *XML/HTML*, pour une analyse ultérieure des résultats d'évaluation de la qualité et de restructuration. Les rapports *XML/HTML* générés contiennent des informations concernant les résultats de prédiction de la qualité et de restructuration ou des suggestions des alternatives de conception.

Le traitement de l'outil pourra se résumer dans « l'algorithme » ci-dessous :

[18] Cette option est disponible seulement dans la partie classique de l'outil.

Pour chaque **classe** de **l'application**

Pour chaque **classification/conclusion finale**

Trouver l'ensemble des **règles** qui concluent sur cette conclusion //ensemble ER

SI il n'y a plus de règles non déclenchée dans ER **ALORS**

Chercher la valeur de conclusion maximale

Donner la valeur maximale trouvée à la classification

Trouver la (es) **métrique(s)** dont les valeurs sont identiques à celle de la classification

Proposer la (es) **restructuration(s)** susceptible de changer les valeurs de métrique(s)

SINON // pour chaque règle dans ER,

(1) SI il n'y a plus de **prémisses** à explorer pour savoir le changement possible **ALORS**

Calculer le min de cet ensemble de valeurs de prémisses,

Déclenché la règle & **Affecter** ce min à la conclusion.

SINON

SI la prémisse courante est une métrique **ALORS**

Explorer le changement possible de cette métrique :

- **Applicable**: est-ce que cette restructuration est applicable ?
- **Taille** de changement : une approximation de cette valeur une fois qu'on fait le changement, selon le rôle de la classe

SINON

Trouver les règles qui concluent sur cette **conclusion intermédiaire**

Pour chaque règle trouvée

Réitérer le processus (1) sur chaque prémisse de cette règle,

Chercher la conclusion maximale

Donner la valeur trouvée à la conclusion intermédiaire en cours de traitement,

Ajouter cette conclusion à la base de faits.

Figure 22. L'algorithme qui résume les traitements de l'outil

5.4 L'ENVIRONNEMENT DE RÉALISATION

5.4.1 Le langage de modélisation UML

Nous avons utilisé le langage unifié de modélisation (Unified Modeling Language, UML) pour modéliser les différents aspects de l'outil. Le choix de ce langage a été motivé par sa simplicité d'utilisation et sa capacité de modéliser l'aspect statique aussi bien que dynamique de notre prototype. En effet, UML est très intuitif, simple, ainsi que plus homogène et cohérent que les autres méthodes. L'aspect formel de sa notation limite les ambiguïtés et les incompréhensions. Son aspect visuel facilite la comparaison et l'évaluation de solutions. Son caractère polyvalent et sa souplesse en font un langage universel. UML définit 9 diagrammes pour représenter les différents points de vue de la modélisation : les diagrammes d'activité, les diagrammes de cas d'utilisation, les diagrammes de classes, les diagrammes de collaboration, les diagrammes de déploiement, les diagrammes d'états transitions, les diagrammes d'objet et les diagrammes de séquence.

5.4.2 Le langage Java

L'outil a été implanté en Java, dans l'environnement de développement *JBuilder*. Le choix de ce langage a été guidé par sa qualité désormais reconnue. En effet, Java est un langage de programmation OO, conçu pour répondre aux besoins de développement d'applications dans un contexte hétérogène, distribué, sécurisé et robuste. Ce langage présente les avantages d'être portable sur plusieurs plates-formes grâce aux notions de machine virtuelle (*Java Virtual Machine* ou JVM), orienté objet et multi-threads. Il offre des fonctionnalités intéressantes pour faciliter la distribution dynamique du code et le transfert des objets. Il est devenu le langage de programmation préféré dans une multitude d'applications : programmes de sécurité réseau, traitement de l'image et du multimédia, *Web* côté client et serveur, ou encore les systèmes d'information qui régissent le cœur de l'entreprise.

5.4.3 L'outil de développement JBuilder

Nous avons opté pour implanter notre outil d'utiliser l'environnement de développement *Jbuilder 4*. En effet, *JBuilder* est entièrement réécrit en Java 2 depuis la version 3.5, et plus particulièrement avec les JFC swings. Il supporte les standards Java les plus récents et offre une plate-forme stable et universelle pour le développement d'applications, d'où la portabilité sous d'autres environnements autres que *Windows* tels *Linux*, *Solaris*, etc. D'autre part, *JBuilder* a l'avantage de générer du code pur Java, sans balises ni code propriétaire, donc portable. En outre, cet outil permet de maintenir la qualité tout au long du processus de développement grâce au test des unités. Il permet aussi d'accélérer la programmation et d'éviter les erreurs de syntaxe en utilisant l'éditeur de code extensible et l'expert d'audit de code, *Code Insight*.

5.4.4 Le système DISCOVER

Pour combler la pénurie de documentation des systèmes étudiés, nous avons utilisé l'outil DISCOVER [DIS00][19]. Ce système de ré-ingénierie permet d'analyser des applications écrites en C/C++ et en Java. DISCOVER se compose d'un ensemble d'outils (Develop/set, CM/set, Reengineer/set, Doc/set, Admin/set) et d'applications pour la compréhension et le développement de logiciels de grande taille. Les outils sont destinés surtout aux gestionnaires de projet, à l'architecte du système et aux développeurs. DISCOVER analyse le code source et crée une base de données appelée *Information Model* (IM), relatant les caractéristiques d'un logiciel.

DISCOVER s'exécute dans l'environnement *Solaris* et offre les fonctionnalités suivantes :

i- Organiser le code selon les types d'entités (fichiers, variables, structures, classes, type de données abstraites, ...) présents dans une application (voir figure 23 ci-dessous).

[19] DISCOVER est une marque déposée de Upspring Software.

ii- Fournir des outils pour visualiser et naviguer à travers le code, les diagrammes entités-relations et la hiérarchie de classes de l'application. La structure arborescente d'une application peut être visualisée à tout moment, permettant ainsi la vérification de l'applicabilité d'une restructuration par la comparaison du rôle joué par la classe au sein de cette transformation avec son rôle actuel donné par l'arbre d'héritage.

iii- Faire des requêtes sur les entités présentes dans les programmes sources. Par exemple : lister toutes les super-classe, lister toutes les variables, lister les méthodes satisfaisant à certains critères, ...

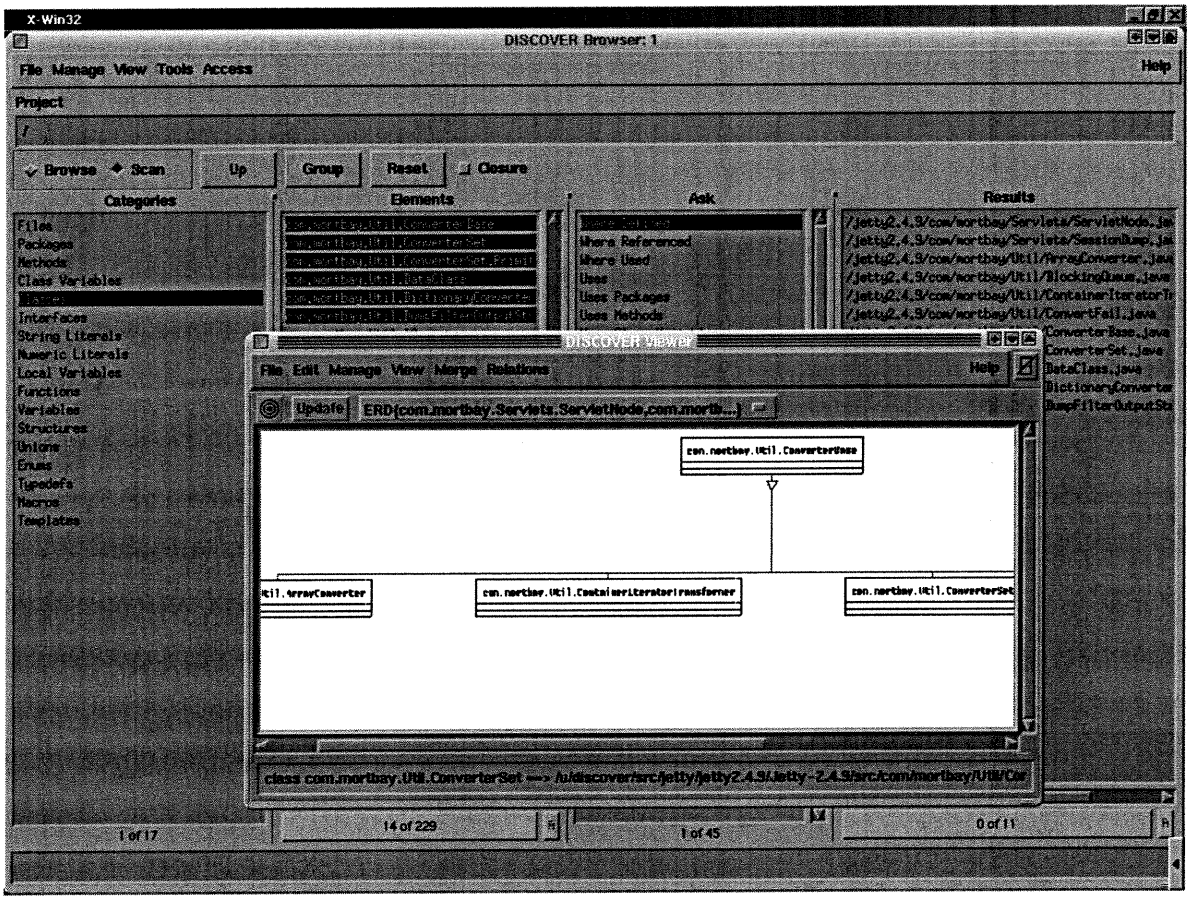


Figure 23. Vue générale de Browser *DISCOVER* et d'une partie de diagramme de classes qu'il génère

5.5 L'IMPLANTATION DE L'OUTIL

5.5.1 Diagramme de classes UML

La figure ci-dessous montre le diagramme de classes de partie floue de l'outil *OO1_Correct*, conçu à moyen du langage unifié de modélisation UML.

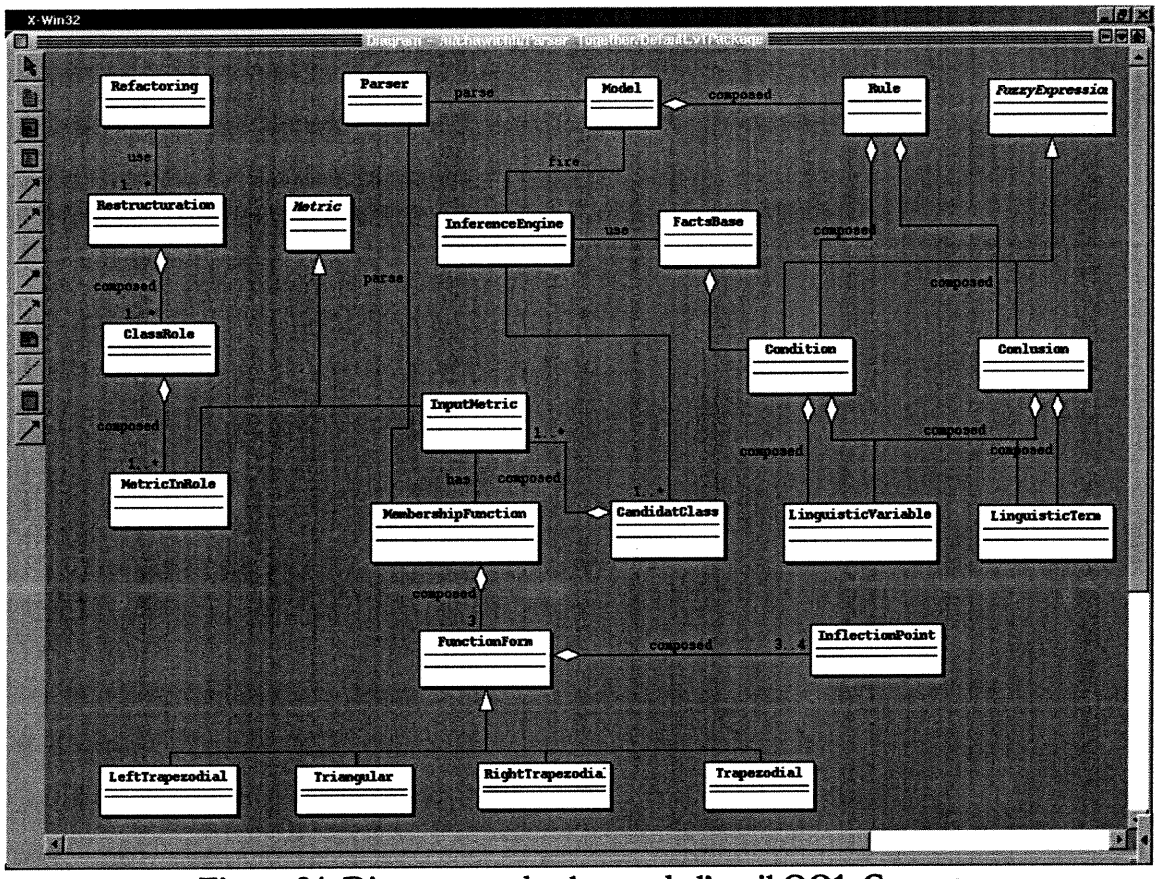


Figure 24. Diagramme de classes de l'outil OO1_Correct

5.5.2 Aperçu rapide des rôles des classes

Dans cette section, nous allons brièvement décrire les rôles joués par les classes dans la hiérarchie de classes ci-dessus.

- Classe *Refactoring* : englobe les méthodes s'occupant des suggestions des alternatives de conception;

- Classe *Restructuration* : représente les restructurations de haut niveau sous forme d'objets. Un objet de type *Restructuration* est constitué d'un ou de plusieurs objets de type *ClassRole*;

- Classe *ClassRole* : représente le rôle d'une classe dans une restructuration. Un objet de type *ClassRole* est composé d'un ou de plusieurs objets de type *MetricInRole*;

- Classe *MetricInRole* : représente les métriques impliquées dans un *ClassRole* d'une restructuration donnée. Elle hérite de la classe abstraite *Metric*;

- Classe *InputMetric* : représente les métriques d'une application. Elle hérite de la classe abstraite *Metric*. Un objet de type *InputMetric* a une fonction d'appartenance. Celle-ci peut prendre plusieurs formes. (Voir la classe *MembershipFunction* ci-après);

- Classe *Metric* : représente les comportements communs (attributs, méthodes) entre les deux classes *MetricInRole* et *InputMetric*. Elle est une classe abstraite;

- Classe *MembershipFunction* : représente les fonctions d'appartenance définies sur les métriques d'entrées. Un objet de type *MembershipFunction* peut avoir trois objets de type *FunctionForm*;

- Classe *FunctionForm* : représente les formes de fonctions d'appartenance. Un objet de type *FunctionForm* peut être un objet de type *LeftTrapezodial*, *RightTrapezodial*, *Triangular* ou *Trapezodial*. Selon la forme de fonction d'appartenance, cet objet peut avoir trois ou quatre points d'inflexion;

- Classe *LeftTrapezodial* : représente la forme trapèze à gauche d'une fonction d'appartenance;

- Classe *RightTrapezodial* : représente la forme trapèze à droite d'une fonction d'appartenance;

- Classe *Triangular* : représente la forme triangulaire d'une fonction d'appartenance;

- Classe *Trapezodial* : représente la forme trapézoïdale d'une fonction d'appartenance;

- Classe *InflectionPoint* : représente les points sur lesquels la forme d'une fonction d'appartenance change son allure;

- Classe *Parser* : représente la classe s'occupant de l'analyse et de parsing de données d'entrée.

- Classe *Model* : représente l'ensemble de règles de prédiction d'un critère de qualité comme la stabilité, maintenabilité, ...;

- Classe *InferenceEngine* : représente le moteur d'inférence qui interprète les règles d'un modèle de qualité. Cette interprétation provoque la modification (ajout) de base de faits;

- Classe *FactsBase* : représente la base des faits. À l'état initial, cette base contient les faits provenant de parsing des métriques d'entrées. Elle acquiert au fur et à mesure les faits résultant de l'interprétation de règles du modèle de qualité;

- Classe *CandidatClass* : représente les classes candidates à un éventuel processus d'estimation/amélioration de leur qualité. Un objet de type *CandidatClass* est composé d'un ensemble d'objets de type *InputMetric*;

- Classe *FuzzyExpression* : cette classe abstraite représente les comportements communs (attributs, méthodes) entre les deux classes *Condition* et *Conclusion*. Elle est surtout utilisée dans le cas où une conclusion intermédiaire figure dans la partie condition d'une règle;

- Classe *Rule* : représente les règles d'un modèle de qualité. Un objet de type *Rule* est composé d'un ensemble (ou d'une liste) d'objets de type *Condition* et d'un seul objet de type *Conclusion*.

- Classe *Condition* : représente les conditions d'une règle de prédiction. Un objet de type *Condition* est composé d'un objet de type *LinguisticVariable* et d'un autre de type *LinguisticTerm*.

- Classe *Conclusion* : représente les conclusions des règles de prédiction. Un objet de type *Conclusion* est composé d'un objet de type *LinguisticVariable* et d'un autre de type LinguisticTerm.

- Classe *LinguisticVariable* : représente le nom d'une métrique, dans le cas d'une condition, ou le nom d'une conclusion intermédiaire/finale, dans le cas d'une conclusion.

- Classe *LinguisticTerm* : représente l'attribut qualitatif ou l'état d'une métrique ou d'une condition.

5.5.3 Les interfaces de OO1_Correct

La figure 25 montre l'interface principale de l'outil. Cette interface est divisée en quatre parties ou sous interfaces. Les sections subséquentes détails l'intérêt de ces sous interfaces

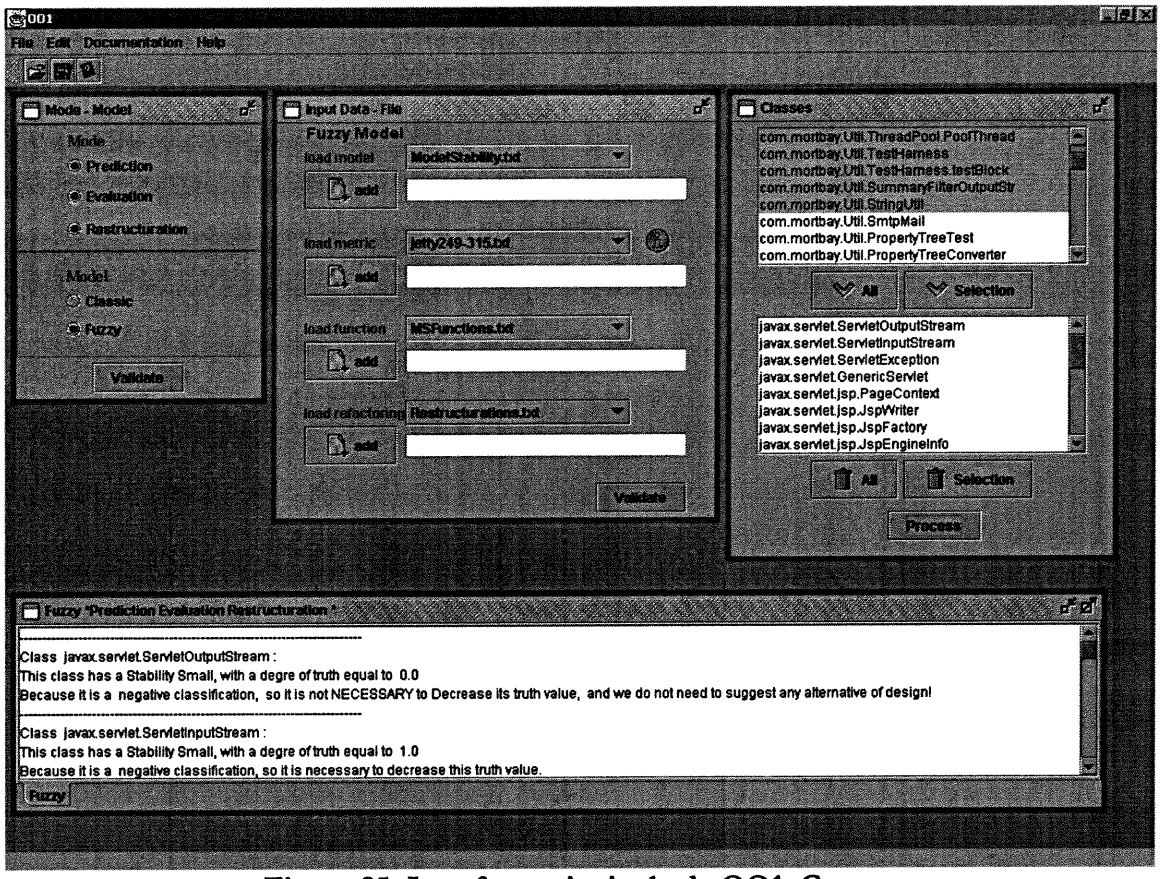


Figure 25. Interface principale de OO1_Correct

5.5.3.1 L'interface « Mode – Model »

Cette interface est divisée en deux parties. La première s'intéresse au mode des traitements de l'outil. En mode Prédiction, l'outil permet d'estimer ou de prédire la qualité d'une classe. En mode Évaluation, l'outil permet d'évaluer l'exactitude ("*accuracy*") d'un modèle de prédiction, en comparant les qualités des classes qu'il en donne avec celles fournies par l'utilisateur. En mode Restructuration, l'outil permet de suggérer des restructurations comme remède aux classes défectueuses.

La deuxième partie de l'interface permet de choisir entre les modèles de prédiction classique et floue.

5.5.3.2 L'interface « Input – Data »

Cette interface fournit à l'utilisateur le moyen pour communiquer les données d'entrée nécessaires à l'outil (modèle de prédiction, valeurs de métriques, fonctions d'appartenance, restructurations) pour se servir lors du traitement. Elle offre les possibilités de choisir les données parmi celles qui sont déjà fournies ou/et d'acquérir de nouvelles données. Ces derniers s'ajoutent à celles qui sont déjà existantes, et elles seront utilisées dans des traitements futurs.

5.5.3.3 L'interface « Classes »

Une fois que les choix de données d'entrée ont lieu, les classes du système à étudier seront affichées dans la partie supérieure de l'interface Classes, permettant à l'utilisateur de sélectionner celles qui désirent traiter. L'utilisateur aura le choix de sélectionner toutes ou une partie de ces classes. Celles qui ont été choisies s'affichent dans la partie inférieure de cette interface. En tout temps, l'utilisateur peut retirer ou ajouter des classes de la liste de classes sélectionnées.

5.5.3.4 L'interface « Fuzzy Prediction Evaluation Restructuration »

Une fois les classes à traiter sélectionnées, l'outil peut entreprendre le traitement; les résultats de prédiction, d'évaluation et de restructuration seront affichés dans cette interface, permettant à l'utilisateur de les visualiser.

Chapitre 6

ÉTUDE DE CAS

Ce chapitre porte sur la validation de notre approche au moyen d'une étude expérimentale, menée sur les classes du système *Jetty*. Dans un premier temps, nous donnerons le modèle de prédiction flou utilisé dans notre étude. Dans un second temps, nous allons procéder à la présentation et à l'analyse des résultats obtenus.

Bien que OO1_correct automatise les tâches de prédiction et de suggestion des restructurations selon l'approche classique et floue, notre étude de cas se restreint cependant à l'approche floue.

6.1 Le modèle prédictif flou utilisé

L'utilité d'un modèle de prédiction est multiple. Ce dernier peut servir à déterminer quelles classes doivent être particulièrement testées et inspectées. Dans le cas de maintenance préventive, il peut déterminer quelles classes doivent être réécrites et de quelle manière. Il peut, enfin, être utilisé pour détecter les composants potentiellement réutilisables dans une application écrite, sans souci de réutilisation.

Le modèle de prédiction de la stabilité utilisé dans notre étude est tiré de [SAR01]. Ce modèle se base sur les métriques de stabilité, présentées au Chapitre 3. Ce modèle est obtenu par l'enrichissement d'un modèle naïf de départ avec des heuristiques du domaine, en rendant plus mature les relations de causalité dans les règles de départ et en utilisant la logique floue pour fuzzyfier les entrées [SAR01].

La figure 26 de la page suivante, montre le modèle de prédiction floue, utilisé dans notre étude expérimentale.

Rule1
IF *DAM is MEDIUM* THEN *potential_coupling is LARGE*

Rule2
IF *DAM is SMALL* THEN *potential_coupling is LARGE*

Rule3
IF *potential_coupling is LARGE* THEN *negative_change_effect is LARGE*

Rule4
IF *negative_change_effect is LARGE* THEN *Stability is SMALL*

Rule5
IF *NAM is LARGE* and *DIT is MEDIUM* THEN *specialization_degree is LARGE*

Rule6
IF *DIT is LARGE* THEN *specialization_degree is LARGE*

Rule7
IF *NAM is LARGE* and *DIT is MEDIUM* and *NOP is SMALL* THEN *potential_specialization is SMALL*

Rule8
IF *DIT is LARGE* and *NOP is SMALL* THEN *potential_specialization is SMALL*

Rule9
IF *potential_specialization is SMALL* and *specialization_degree is LARGE* THEN *modification_ difficulty is LARGE*

Rule10
IF *modification_ difficulty is LARGE* THEN *Stability is SMALL*

Rule11
IF *OCAEC is LARGE* THEN *coupling_impact is LARGE*

Rule12
IF *OCAEC is MEDIUM* and *NAA is SMALL* THEN *attribute_change_effect is LARGE*

Rule13
IF *attribute_change_effect is LARGE* THEN *negative_change_effect is LARGE*

Rule14
IF *negative_change_effect is LARGE* THEN *Stability is SMALL*

Figure 26. Le modèle prédictif flou utilisé

6.2 Des exemples de prédictions

Les classes de l'application *Jetty* constitueront la matière première de notre étude expérimentale. Il s'agit d'un serveur *Web* écrit entièrement en Java et comptant 228 classes. La raison pour laquelle ce système a été choisi réside dans la disponibilité de plusieurs versions de cette application[20], ce qui facilite l'évaluation de la stabilité. En outre, il sera fort intéressant de tester les règles de prédiction sur les systèmes utilisés pour élaborer ces règles, et dont *Jetty* fait partie.

Pour les classes de *Jetty*, l'outil a détecté la présence de 148 d'entre elles ayant une tendance à être instable. En regardant le diagramme des classes de l'application et son modèle entité-relation, les deux obtenus au moyen du système DISCOVER, et en inspectant le code source des classes impliquées, nous avons pu tirer les conclusions suivantes :

i- 105 suggestions de restructuration proposées (ou 70.94 %) pourront être appliquées dans le contexte de l'application.

ii- 43 suggestions restantes (29.06 %) ne sont pas applicables dans le contexte de l'application étudiée.

Des exemples des restructurations applicables et d'autres qui sont inapplicables seront donnés dans les sections subséquentes. Le tableau ci-dessous contient les valeurs de métriques de classes intervenant dans ces exemples.

Nom de la classe	**Métriques**								
	OCAEC	NOP	DIT	NAM	NAA	DAM	NAI	NTA	NPA
Table	0	0	2	28	8	0.4	1	3	3
TestTable	0	0	3	30	18	0	3	4	4
MultiTestTable	0	0	3	31	13	0	3	3	3
TestHarness	0	0	1	2	0	0	2	1	4

Tableau 39. Valeurs des métriques des classes impliquées dans les exemples

[20] Nous disposons de dix versions consécutives du serveur *Jetty*. Nous nous sommes basés sur les deux dernières versions de ce système dans notre étude de cas.

6.2.1 Exemple d'une restructuration applicable

La figure 27 reproduit le résultat d'une exécution du OO1_correct où la requête est une demande de prédiction de stabilité et de suggestions de restructurations pour les classes *Table*, *TestTable* et *MultiTestTable*. Ces classes sont évaluées par l'outil comme ayant une tendance à une stabilité faible avec un degré de vérité égal à 1 (ou selon le formalisme de la logique floue : *Stability is Small with truth value equal to 1*), c'est-à-dire qu'on est certain que les classes auront une stabilité très faible.

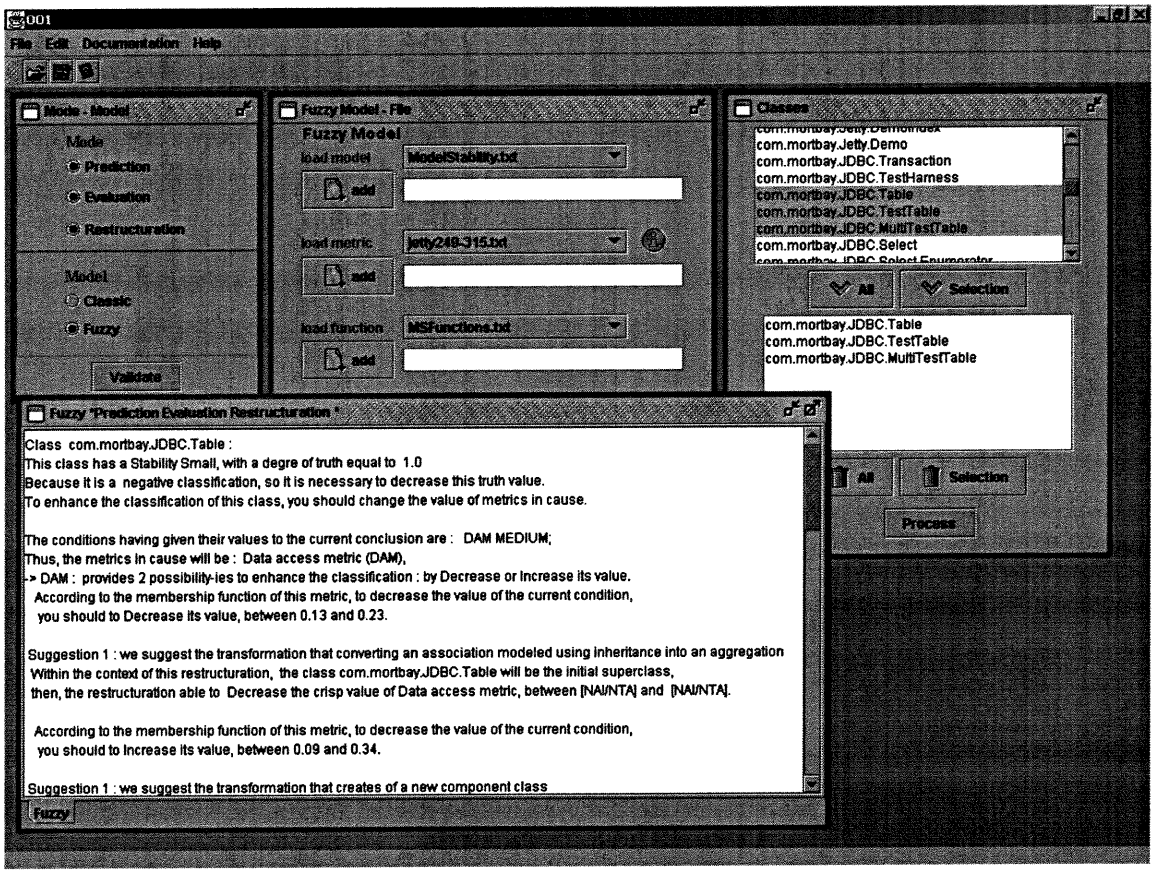


Figure 27. La suggestion des alternatives de conception pour la classe *Table*

Pour la classe *Table*, l'outil a détecté que la métrique *DAM,* dérivant de la prémisse *DAM MEDIUM,* est la cause de son faible niveau de stabilité. Si nous désirons améliorer la stabilité de cette classe, nous devons faire diminuer la valeur de vérité de la prémisse

DAM MEDIUM. D'après la fonction d'appartenance définie sur *DAM* (voir ANNEXE), la diminution de la valeur de cette prémisse peut être obtenue dans deux cas différents : i) soit en diminuant *DAM* d'au moins une valeur comprise entre 0.13 et 0.23; et ii) soit en augmentant *DAM* d'au moins une valeur comprise entre 0.09 et 0.34.

Pour le premier cas, l'outil propose la restructuration qui convertit la relation d'héritage, joignant cette classe à un ou plusieurs de ses sous-classes, en une relation d'agréation. Cette restructuration est retenue pour deux simples et très bonnes raisons. Premièrement, elle apporte une diminution convenable de la valeur concrète de l'ordre *NAI/NTA* ou *1/3 = 0.33*. De plus, le rôle joué par cette classe dans la hiérarchie de classes (voir figure 28) est identique à celui joué au sein de la restructuration. Deuxièmement, et comme nous allons le voir sous peu, cette restructuration convient aussi aux deux classes *TestTable* et *MultiTestTable,* du point de vue de la modification qu'elle apporte aux valeurs de métriques et au rôle joué par ces classes.

Pour le deuxième cas, l'outil propose deux restructurations susceptibles d'apporter une augmentation à la valeur concrète de *DAM* : la création d'une classe composante à partir des éléments de *Table* ou bien la conversion de la relation d'héritage en une agrégation. Ces deux restructurations sont abandonnées puisque l'augmentation qu'elles apportent à *DAM*, soit $[NPA / (NTA) \times (NTA+1)]$ ou $1/ (3) \times (3+1) = 0.08$, est plus petite que la valeur minimale requise.

Par ailleurs, l'outil a détecté pour les deux classes qui restent que la métrique *DAM*, en provenance de la prémisse *DAM SMALL*, est la cause de leur faible degré de stabilité. Améliorer la stabilité de ces deux classes revient à diminuer le degré de vérité de la prémisse *DAM SMALL*. Selon la fonction d'appartenance définie sur *DAM*, cette diminution se fait sentir graduellement dès que la valeur concrète de cette métrique franchit le seuil de 0.17. L'outil propose deux suggestions différentes pour obtenir cette variation (voir figure 29) : la création d'une classe composante ou bien la conversion de la relation d'héritage en une agrégation.
La première suggestion est abandonnée, étant donné que les deux classes sont très petites et qu'il n'y a pas assez de membres pour former une classe composante.

La deuxième restructuration suggérée consiste à la transformation de la relation d'héritage joignant ces deux classes avec leur super-classe. Cette suggestion est semble applicable puisque les deux classes en question ont déjà *Table* comme super-classe (voir figure 28) et que la variation qu'elle apporte à la valeur de *DAM* est de [*NPA* / (*NTA*) x (*NTA*+1)], soit respectivement *1/5* et *1/4* pour *TestTable* et *MultiTestTable*. Ces deux valeurs appartiennent chacune à l'intervalle demandé, qui est compris entre 0.17 et 1.26.

On conclut qu'au lieu d'appliquer trois restructurations sur ces trois classes, il suffit d'appliquer une seule restructuration, soit la conversion d'une relation d'héritage en une agrégation, pour améliorer la stabilité des trois classes en question.

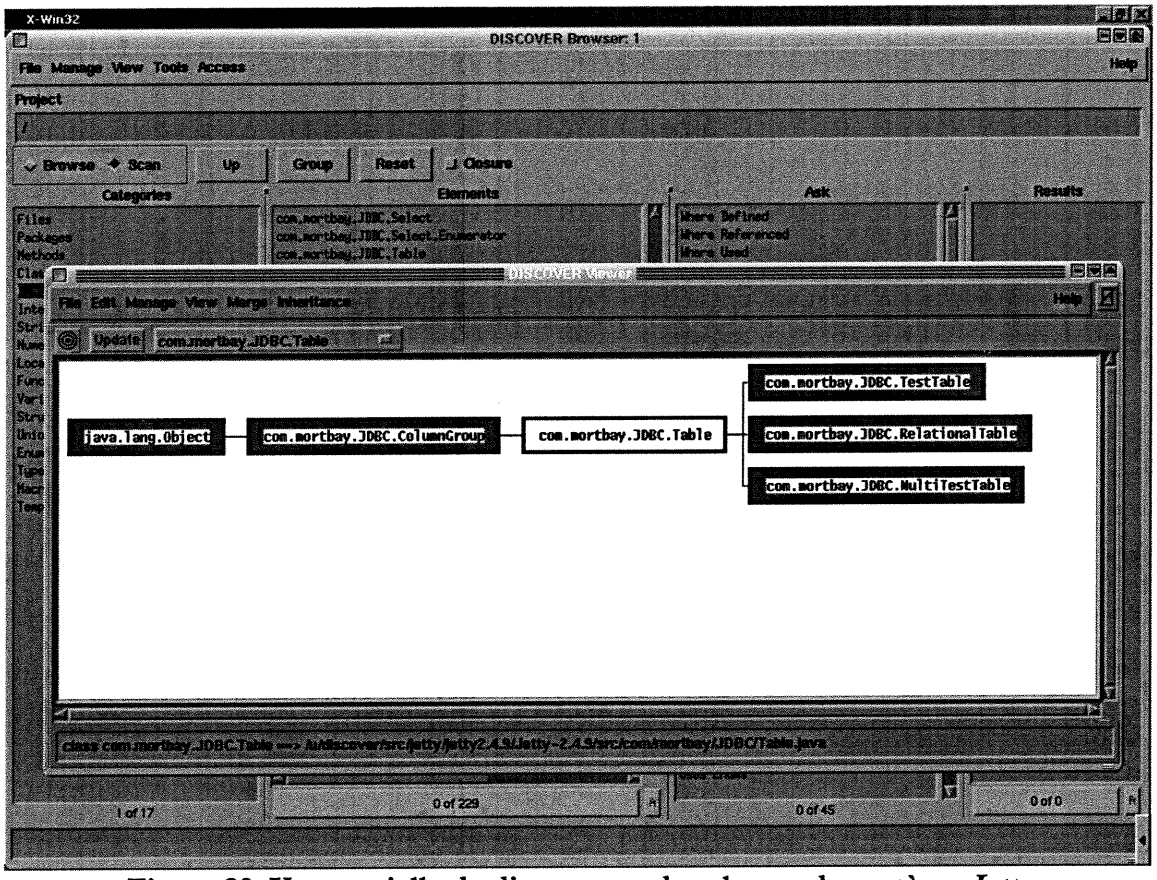


Figure 28. Vue partielle de diagramme des classes du système *Jetty*

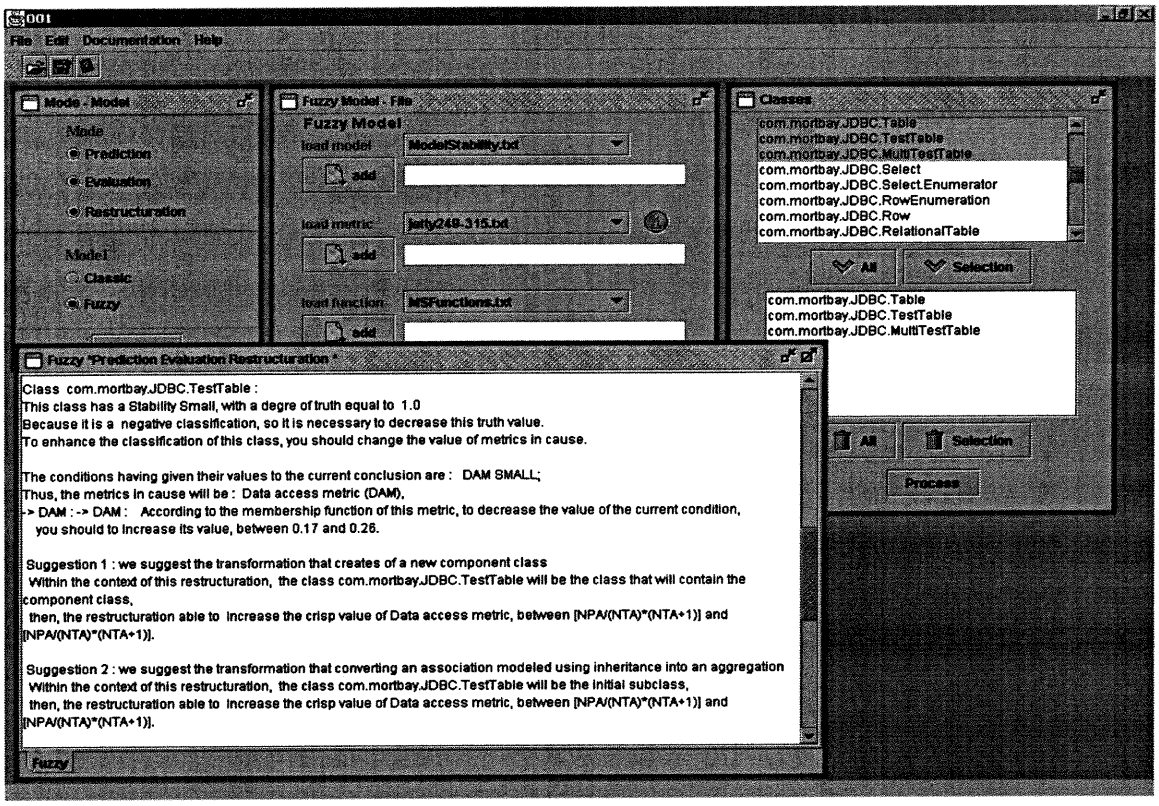


Figure 29. Les restructurations suggérées par l'outil pour les deux classes *TestTable* et *MultiTestTable*

6.2.2 Exemple d'une restructuration non applicable

Nous présentons dans cette section l'exemple de restructurations suggérées par l'outil, mais qui ne sont pas applicables au contexte des classes en question.

La figure 30 ci-dessous reproduit les résultats d'évaluation de la stabilité de la classe *TestHarness* ainsi que les restructurations proposées par l'outil. La classe est évaluée comme ayant une tendance à être instable, avec un degré de vérité égal à 1. L'outil a détecté que la métrique *DAM,* découlant de la prémisse *DAM SMALL*, est la cause de cette mauvaise qualité. Pour améliorer la stabilité de cette classe, nous devons faire diminuer la valeur de vérité de prémisse *DAM SMALL*. Selon la fonction d'appartenance

définie sur *DAM* (voir ANNEXE), diminuer la valeur de vérité de cette prémisse revient à augmenter la valeur concrète de la métrique *DAM* d'au moins 0.17.

Deux restructurations ont été sélectionnées à ce propos. La première d'entre elles consiste à la création d'une classe composante, à partir des membres de classe *TestHarness*. Après une inspection rigoureuse du code source (cf. figure 31), nous avons constaté que la classe ne contient pas un nombre suffisant d'attributs et de méthodes pour créer une classe composante. Par conséquent, cette restructuration ne pourra être appliquée.

La deuxième proposition consiste à la conversion de la relation d'héritage, reliant *TestHarness* à sa super-classe, en une relation d'agrégation. *TestHarness* ne possède pas une super-classe définie dans l'application (voir figure 31). Ainsi, cette restructuration ne semble pas applicable au contexte de classe en question.

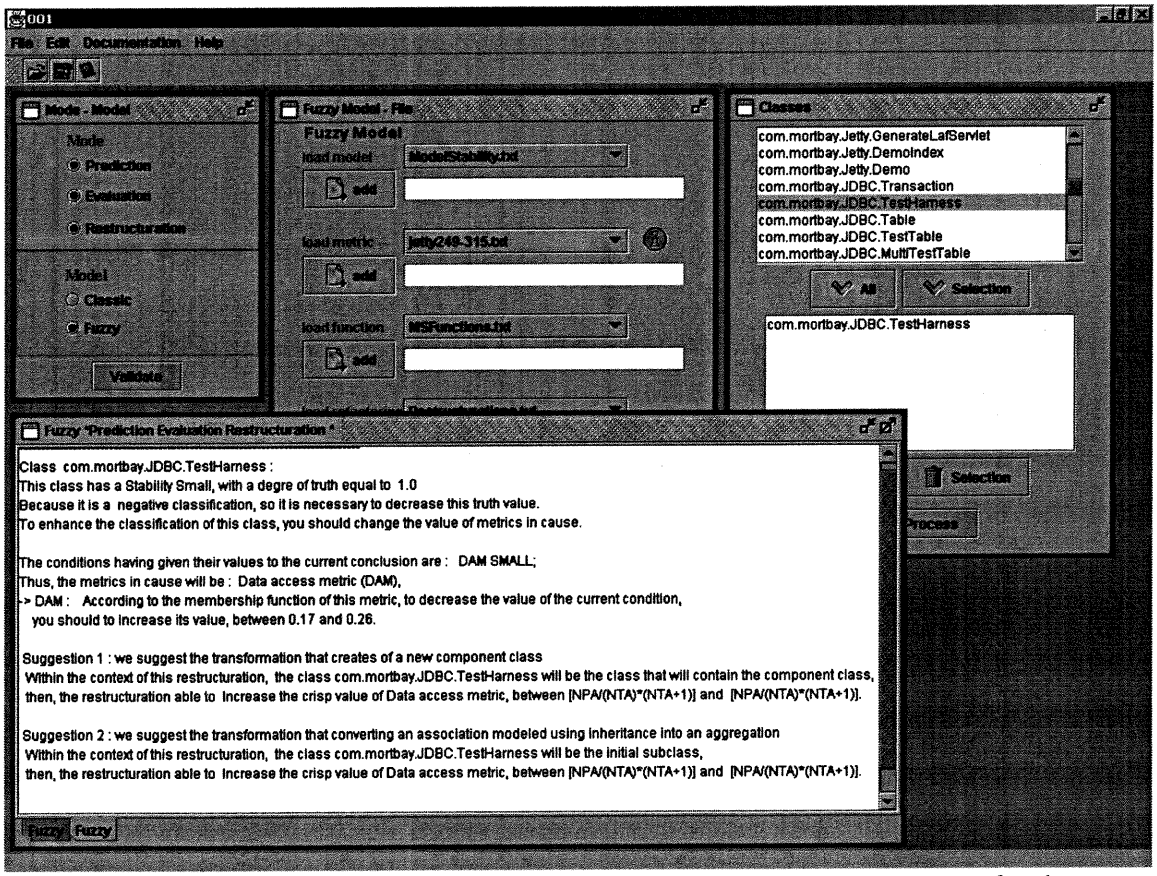


Figure 30. Exemple d'une restructuration non applicable au contexte de classe *TestHarness*

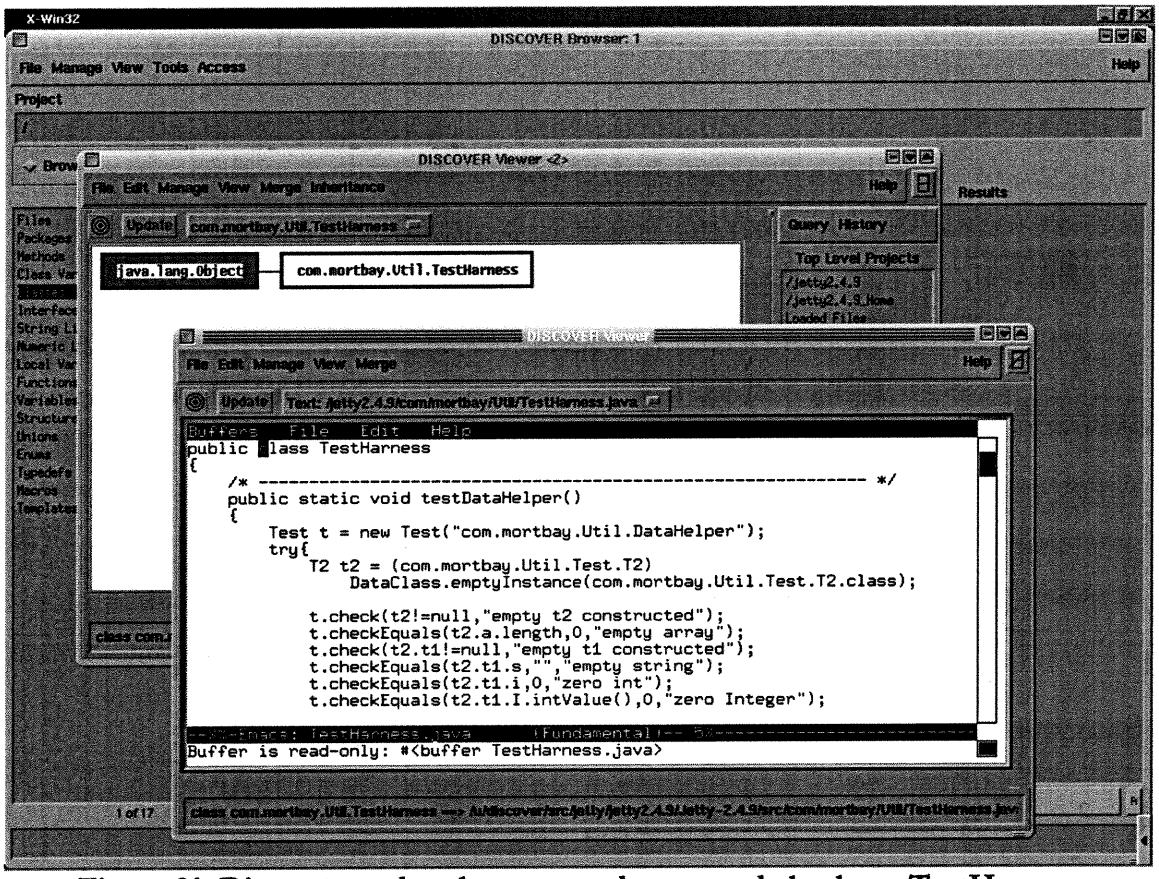


Figure 31. Diagramme des classes et code source de la classe TestHarness

6.3 DISCUSSION DES RÉSULTATS

L'analyse des résultats de prédiction nous montre que la plupart des classes dont la valeur de métrique *DAM* est égale à zéro (ou presque) ont été jugées par l'outil comme des classes instables. Ceci nous indique que nos mesures peuvent prédirent, avec un niveau élevé d'exactitude, quelles classes peuvent être instables. En effet, rappelons que la stabilité d'une classe est définie comme étant la stabilité de son interface vis à vis des changements subséquents ; et que *DAM* représente la proportion d'attributs privés et protégés du nombre total d'attributs dans une classe. Le fait que la valeur de métrique *DAM* est nulle signifie que les attributs privés et protégés ont disparu de l'interface des classes, ce qui explique que les classes aient une tendance à être instable.

En ce qui concerne les suggestions d'alternatives, la majorité des restructurations suggérées était la création d'une classe composante et/ou conversion d'une relation d'héritage en agrégation. La raison pour laquelle ces deux restructurations ont été choisies réside dans leur capacité à modifier la valeur de la métrique, *DAM*, cause de l'instabilité des classes. En effet, l'extraction des données membres d'une classe pour créer une classe composante diminue le nombre d'attributs et des méthodes de la classe concernée. Par conséquent, la valeur de *DAM* sera diminuer. D'autre part, la conversion de la relation d'héritage en agrégation entraîne le déplacement d'un certain nombre d'attributs et de méthodes d'une super-classe vers une de ses sous-classes, ce qui aura pour effet de réduire la valeur de la métrique *DAM* de la super-classe. En outre, cette opération contribuera à l'augmentation de la valeur de *DAM* pour la classe sous-classe.

La vérification de l'applicabilité d'une transformation a été restreinte à une confrontation simple de rôle de classe au sein d'une restructuration proposée à celui de diagramme de classes. Nous nous sommes limités à une telle vérification car la seule source d'information avec laquelle nous avons travaillé est le code source de l'application. Une vérification plus poussée semble toujours nécessaire pour assurer l'efficacité de notre approche.

Chapitre 7

CONCLUSION

7.1 Synthèse

Dans le but d'améliorer le logiciel existant ainsi que sa compréhension, il est nécessaire de fournir des informations sur les défauts existants et d'offrir des guides pour y remédier. Dans les systèmes à objets, les modèles de prédiction utilisant des métriques de conception peuvent servir à détecter les classes qui ont une propension à générer des erreurs. La détection proprement dite ne pourra être utile tant qu'elle ne suggère pas des alternatives de conception pour balayer les défectuosités trouvées. D'autre part, l'utilisation des restructurations élémentaires ou complexes, sans tenir compte de leur impact sur les différentes composantes du système, peut dégrader l'intérêt pour lequel elles avaient été utilisées.

C'est dans l'optique de rapprocher les techniques d'estimation de la qualité de celles de la restructuration de logiciels qu'a été effectué le travail décrit dans ce mémoire. L'approche que nous avons présentée permet à la fois la détection et la correction automatique des défauts de conception, en s'appuyant sur des modèles d'estimation de la qualité, écrits en logique floue, et des restructurations.

Dans un premier temps, nous avons évalué l'impact de neuf restructurations de bas niveau et de quatre de haut niveau sur les valeurs d'une vingtaine de métriques de la stabilité. Suite à cette étape, une liste de catalogues qui imputent les changements de valeurs de ces métriques à telle ou telle restructuration a été créée.

Nous avons analysé, dans un deuxième temps, la possibilité de changement du degré d'appartenance d'une métrique à un ensemble flou donné, selon la valeur concrète de cette métrique et les formes de fonction d'appartenance définies sur celle-ci.

Le choix d'une restructuration pour une situation symptomatique se base sur la direction et l'ampleur de changement qu'une métrique qui a causé l'anomalie doit avoir. Il se base aussi sur les possibilités de variation offertes par la fonction d'appartenance définie sur cette métrique et sur la modification que la restructuration en question apporte à cette métrique.

Notre approche a été mise en œuvre à l'aide de l'outil OO1_correct, permettant à la fois d'évaluer la stabilité d'une classe donnée et de suggérer des transformations pour en améliorer la stabilité. Le comportement de cet outil est similaire à celui d'un correcteur grammatical de texte : il se contente de la détection des anomalies et de suggestion de restructurations pour les corriger et il laisse à l'utilisateur la décision finale d'accepter ou de refuser les modifications proposées.

Notre technique a été validée sur un ensemble de classes (au nombre de 228), écrites entièrement en Java. Les transformations proposées, en utilisant des métriques de stabilité, étaient généralement faisables et pertinentes en matière de rôle joué par les classes étudiées, au sein des restructurations proposées et dans l'arbre d'héritage. Une vérification plus approfondie s'avère difficile vu l'absence de documentation du système étudié.

Bien que nous nous soyons intéressés, dans ce travail, à la prédiction et à l'amélioration de la stabilité du système à objets, cette technique reste également valide pour tout autre critère de la qualité (maintenabilité, réutilisabilité, ...) et s'applique aussi à tout autre modèle de prédiction, comme le modèle de régression linéaire, les réseaux bayésiens, etc.

7.2 Travaux Futurs

Pour ce qui est des limites de notre prototype, elles inspirent une série de futurs travaux. En proposant une transformation quelconque, l'outil se base sur des critères assez simples, comme la comparaison de l'ampleur de changement qu'une restructuration provoquera avec celui demandé par les fonctions d'appartenance. Une telle comparaison peut réduire considérablement la pertinence des transformations suggérées, puisqu'elle ne tient pas compte du contexte du système. Ainsi, d'autres critères beaucoup plus précis s'avèrent nécessaires. Par exemple, si pour une classe donnée la transformation proposée est la conversion d'une relation d'héritage en une agrégation, dans laquelle cette classe joue le rôle de super-classe, il sera nécessaire de vérifier, avant de proposer cette transformation à l'utilisateur, que la classe en question ait déjà une relation d'héritage et possède effectivement des descendants. Cette analyse contextuelle fait appel au diagramme de classes du système et elle peut être automatisée et facilement incorporée dans l'outil, ce qui augmente la précision des résultats.

Une autre amélioration serait l'enrichissement des restructurations déjà existantes. Tel qu'il est montré au chapitre 5, les restructurations proposées se sont limitées à l'extraction d'une classe composante à partir des membres d'une classe, ou alors à la conversion d'une relation d'héritage en une agrégation. Ces deux restructurations ont été utilisées excessivement vu leur capacité à améliorer la valeur de la métrique *DAM*, sans justifier pour autant leur utilisation. Par exemple, le passage d'une relation d'héritage à une agrégation n'est pas toujours évident, puisqu'il touche au plus profond la conception du système, ce qui provoque une hésitation chez l'utilisateur avant de l'appliquer.

Un autre chemin à explorer serait celui de l'utilisation de patrons de conception (*Design Pattern*) au lieu de *refactoring*, lors de la suggestion des alternatifs de conception. Un travail dans ce sens est en cours.

Il reste à mentionner que l'outil entant que tel est ouvert à toute éventualité d'amélioration et de perfectionnement futur. Il consiste par exemple à l'amélioration de performance et de temps d'exécution (décentralisation de traitements et les échelonner sur les étapes

appropriées); à l'extension des fonctionnalités de *Parser* pour qu'il prenne en compte les diagrammes de classes des systèmes étudiés; et enfin à l'utilisation des Applet et Servelts pour rendre l'outil accessible à tout le monde via le *Web*.

Actuellement, l'outil OO1_correct fait l'objet de plusieurs travaux de perfection, dans le cadre de projet des cours en génie logiciel.

Bibliographie

[BAN87] Banerjee J. & Kim W.: "Semantics and implementation of schema evolution in object-oriented databases". *In Proceedings of the ACM SIGMOD Conference*, 1987.

[BAS96] Basili V., Briand L. & Melo W.: "A Validation of Object-Oriented Design Metrics as Quality Indicators". *IEEE Transaction on Software Engineering*, vol. 22, no. 10, 1996 pp-751-761.

[BOU02] Boukadoum M., Chawiche H. M., Sahraoui H. A., Mai G. & Sarhani M. A: "Extension of rule-based prediction systems for decision making support and refactoring of OO software". *Soumis à International Conference on Software Maintenance*; 2002.

[BOU01] Bouchard P. : "*Conception de Systèmes Experts d'Identification de Cibles à l'aide de la Logique Floue*". Thèse de Maîtrise, Université de Montréal, DMS, 2001.

[BOU00] Bouchon B., Yager R. & Zadeh L. A: "*Information, uncertainty, and fusion*", Kluwer Academic Publishers, Boston, 2000.

[BRI96] Briand L., Daly J. & Wüst J. *A:* "Unified Framework for Coupling Measurement in Object-Oriented Systems". *Rapport Technique ISERN 96-14, Fraunhofer Institute for Experimental Software Engineering, Germany*, 1996.

[BRI97] Briand L., Devanbu P. & Melo W.: "An Investigation into coupling Measures for C++". *In Proceedings Of the 19th Int'l Conf. On Software Engineering*, 1997.

[CAS91] Casais E. : "*Managing Evolution in Object Oriented Environments: An Algorithmic Approach.*" Thèse de Doctorat, Université de Genève, 1991.

[CAS94] Casais E.: "Automatic Reorganization of Object Oriented Hierarchies: a Case Study". *Journal of Object Oriented Systems*, p. 95-115, 1994.

[CHI94] Chidamber S. & Kemerer C.: "A Metrics Suite for Object-Oriented Design". *IEEE Transactions on Software Engineering*, June, p. 476-492, 1994.

[DEI00] Deitel H.M, Deitel P.J.: "*Java: How to Programme*". Third Edition. Prentice Hall, 2000.

[DEM99] Demeyer S. & Ducasse S.: "Metrics, Do they really help?". *In Proceedings of LMO*, 1999.

[DIS00] Discover information set. On-line at www.upspringsoftware.com /products/discover/index.html.

[DUB80] Dubois D. & Prade H.: "*Fuzzy sets and Systems: theory and applications*". Academic Press, Toronto, 1980.

[ELE01] El-Emam K., Melo W. & Machado J.: "The prediction of faulty classes using object-oriented design metrics". *In Journal of Systems and Software*, 56(1): 63-75, 2001.

[FEN91] Fenton N.: "*Software Metrics: A Rigorous Approach*". Chapman & Hall, London, 1991.

[FOW99] Fowler M.: "*Refactoring: Improving the Design of Existing Programs*". Addison Wesley, 1999.

[GAC997] Gacôgne L. : "*Éléments de logique floue*". Hermès, Paris, 1997.

[GAM95] Gamma E., Helm R., Johnson R. & Vlissides J.: "*Design Patterns. Elements of Reusable Object-Oriented Software*". Addison Wesley, 1995.

[JAC97] Jacquet J-P, Abran A. & Dupuis R.: "Une analyse structurée des méthodes de validation de métriques". *Dans les Dixièmes journées internationales «Le génie logiciel et ses applications»* (GL97), Paris, EC2D, 1997.

[LIW93] Li W. & Henry S.: "Object Oriented Metrics that Predict Maintainability". *In Journal of Systems and Software*. Vol.23, No.2, 1993.

[MAO98] Mao Y., Sahraoui H. A. & Lounis H.: "Reusability Hypothesis Verification Using Machine Learning Techniques: A Case Study". *In Proceedings of IEEE Automated Software Engineering Conference,* 1998.

[MIC99a] Miceli T. : "*Amélioration de la qualité des systèmes à objets à l'aide de transformations et de métriques*". Thèse de Maîtrise, Université de Québec à Montréal; 1999.

[MIC99b] Miceli T., Sahraoui H. A. & Godin R., "A Metric Based Technique For Design Flaws Detection And Correction". *Proceedings of IEEE Automated Software Engineering Conference*, 1999.

[MOO96] Moore I.: "Automatic Inheritance Hierarchy Restructuring and Method Refactoring". *In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96), CA, USA: ACM SIGPLAN Notices*, pp. 235-250. 1996.

[OCI99] O'Cinnéide, M. O. & Nixon, P. A.: "Methodology for the Automated Introduction of Design Patterns". *In Proceedings of the IEEE International Conference of Software Maintenance (ICSM'99), Oxford, England*, pp. 463-472. 1999.

[OPD92] Opdyke W. F.: "*Refactoring Object-Oriented Frameworks*". Thèse de doctorat, Université de l'Illinois, 1992.

[OPD93] Opdyke W. F. & Johnson R.: "Creating Abstract Super classes by Refactoring". *In Proceeding of CSC'93: The ACM 1993 Computer Science Conference*, 1993.

[ROB97] Roberts D. B, Brant J. & Johnson R. E.: "A Refactoring Tool For Smalltalk". *TAPOS 3(4)*; pp. 39-42. 1997.

[ROB99] Roberts D. B.: "*Practical Analysis for Refactoring*". Thèse de doctorat, Université de l'Illinois, 1999.

[SAH97] Sahraoui H. A., Lounis H. & Melo W.: "Identifying and Measuring Coupling in OO Systems". *Rapport technique CRIM-97/11-82*, 1997.

[SAH00] Sahraoui H. A., Godin R. & Miceli T.: "Can Metrics Help to Bridge the Gap Between the Improvement of OO Design Quality and Its Automation?" *In Proceedings of International Conference on Software Maintenance*, 2000.

[SAH02] Sahraoui H. A., Boukadoum M., Chawiche H. M., Mai G. & Sarhani M. A.: "A fuzzy logic framework to improve the performance and interpretation of rule-based quality prediction models for object-oriented software". À paraître dans *COMPSAC 2002, Oxford (England), August 2002.*

[SAR01] Sarhani M. A : "*Extension Des Modèles De Prédiction De La Qualité Du Logiciel En Utilisant La logique Floue Et Les Heuristiques Du Domaine*". Thèse de Maîtrise, Université de Montréal, DIRO, 2001.

[TOK99] Tokuda L. & Batory D.: "Evolving Object-Oriented Designs with Refactorings". *Proc. of IEEE Automated Software Engineering Conference*, 1999.

[UNG87] Ungar D. & Smith R. B.: "Self: The Power of Simplicity". *In Proceedings of OOPSLA*, pp. 227-241, Orlando, FL, October 1987.

[VOY87] Voyer R. : "*Moteurs de Systèmes Experts*". Éditions EYROLLES 1987.

[ZAD65] Zadeh. L. A.: "Fuzzy Set". *Information and Control* Vol. 8, pp. 338-353, 1965.

[ZAD94] Zadeh. L. A.: "Soft Computing and Fuzzy Logic", *IEEE Transaction*, 1994.

Annexe

A.1 Les fonctions d'appartenance définies sur les métriques

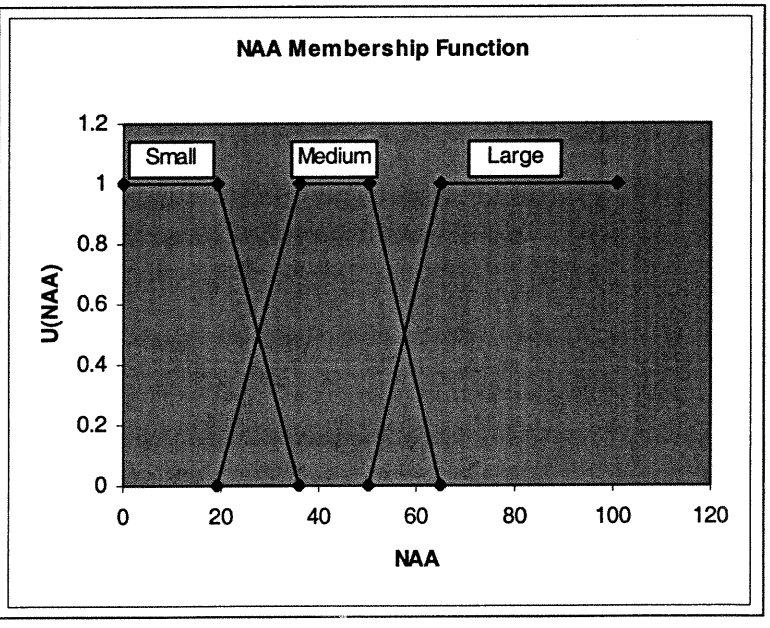
Les fonctions d'appartenance peuvent théoriquement prendre n'importe quelle forme. Toutefois, elles sont souvent définies par des segments de droites, et dites « linéaires par morceaux ». Ces fonctions d'appartenance sont très utilisées car elles sont simples et comportent des points permettant de définir les zones où la notion est vraie, les zones où elle est fausse.

Ce sont des fonctions d'appartenance de ce type qui sont utilisées dans la suite de ce document.

Dans ce qui suit sont les fonctions d'appartenance définies sur les six métriques de la stabilité. Ces fonctions sont tirées de [SAR01].

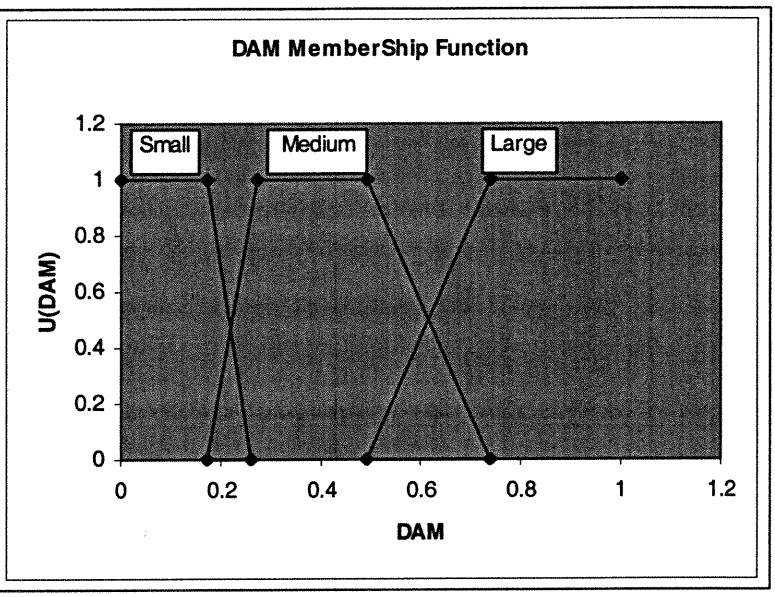
A.1.1 La fonction d'appartenance définie sur NAA

Metric Value	Fuzzy Value
0	1
19	1
36	0
19	0
36	1
50	1
65	0
50	0
65	1
101	1



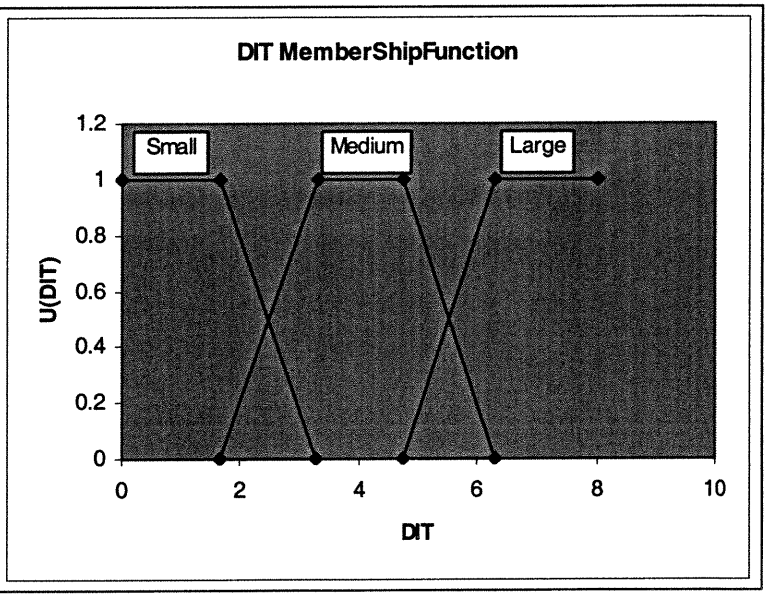
A.1.2 La fonction d'appartenance définie sur DAM

Metric Value	Metric Value
0	1
0.17	1
0.26	0
0.17	0
0.27	1
0.49	1
0.74	0
0.49	0
0.74	1
1	1



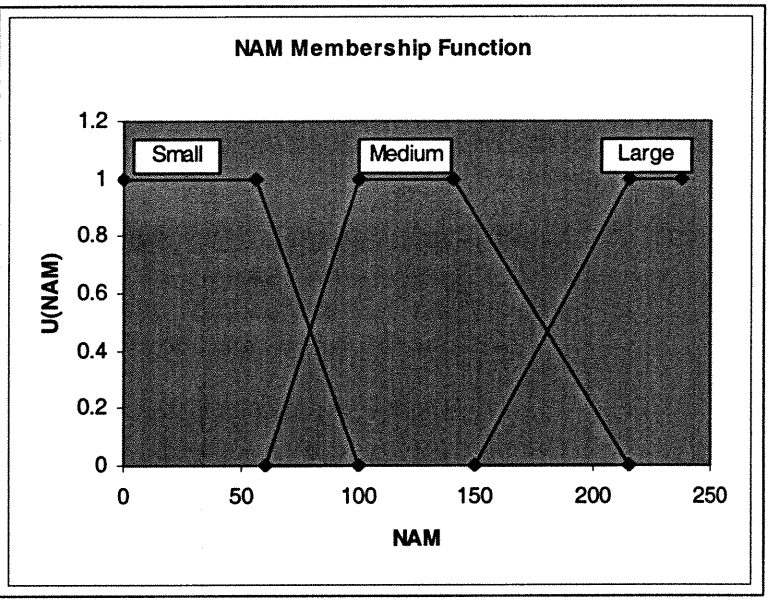
A.1.3 La fonction d'appartenance définie sur DIT

Metric Value	Fuzzy Value
0	1
1.65	1
3.25	0
1.65	0
3.3	1
4.75	1
6.3	0
4.75	0
6.3	1
8	1



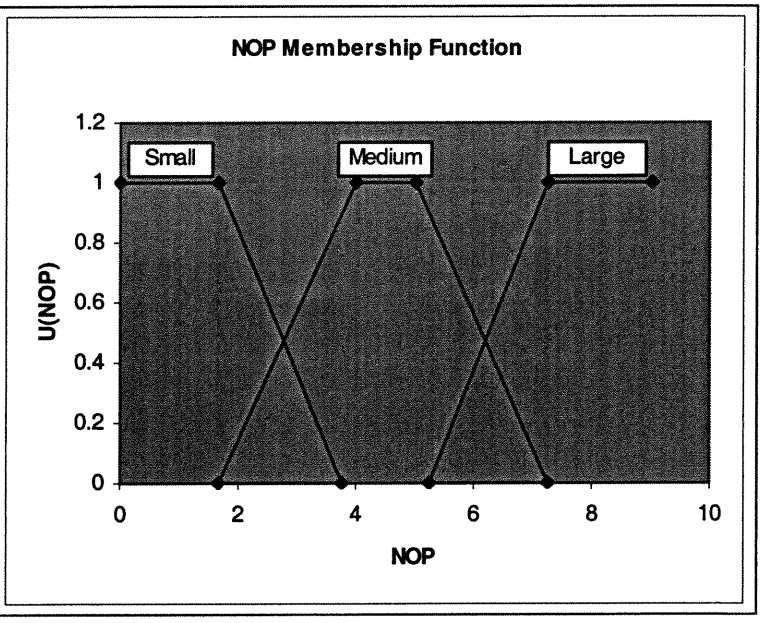
A.1.4 La fonction d'appartenance définie sur NAM

Metric value	Fuzzy value
0	1
56	1
100	0
60	0
100	1
140	1
215	0
150	0
215	1
237	1



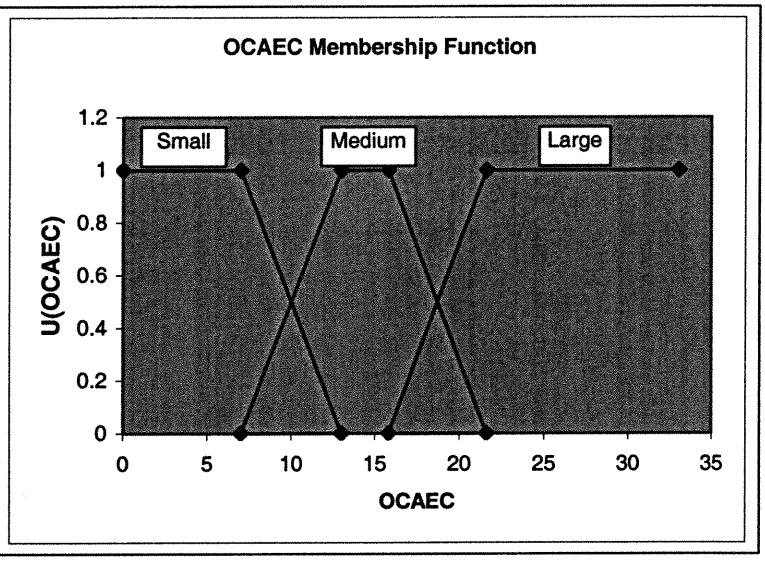
A.1.5 La fonction d'appartenance définie sur NOP

Metric Value	Fuzzy Value
0	1
1.65	1
3.75	0
1.65	0
4	1
5	1
7.25	0
5.25	0
7.25	1
9	1



A.1.6 La fonction d'appartenance définie sur OCAEC

Metric Value	Fuzzy Value
0	1
7	1
13	0
7	0
13	1
15.8	1
21.6	0
15.8	0
21.6	1
33	1



A.2 Les fonctions d'appartenance (en format texte)

Métrique **DAM**
LeftTrapezodial; SMALL; 0; 1; 0.17; 1; 0.26; 0
Trapezodial; MEDIUM; 0.17; 0; 0.27; 1; 0.49; 1; 0.74; 0
RightTrapezodial; LARGE; 0.49; 0; 0.74; 1; 1; 1

Métrique **OCAEC**
LeftTrapezodial; SMALL; 0; 1; 7; 1; 13; 0
Trapezodial; MEDIUM; 7; 0; 13; 1; 15.8; 1; 21.6; 0
RightTrapezodial; LARGE; 15.8; 0; 21.6; 1; 33; 1

Métrique **DIT**
LeftTrapezodial; SMALL; 0; 1; 1.65; 1; 3.25; 0
Trapezodial; MEDIUM; 1.65; 0; 3.3; 1; 4.75; 1; 6.3; 0
RightTrapezodial; LARGE; 4.75; 0; 6.3; 1; 8; 1

Métrique **NOP**
LeftTrapezodial; SMALL; 0; 1; 1.65; 1; 3.75; 0
Trapezodial; MEDIUM; 1.65; 0; 4; 1; 5; 1; 7.25; 0
RightTrapezodial; LARGE; 5.25; 0; 7.25; 1; 9; 1

Métrique **NAM**
LeftTrapezodial; SMALL; 0; 1; 56; 1; 100; 0
Trapezodial; MEDIUM; 60; 0; 100; 1; 140; 1; 215; 0
RightTrapezodial; LARGE; 150; 0; 215; 1; 237; 1

Métrique **NAA**
LeftTrapezodial; SMALL; 0; 1; 19; 1; 36; 0
Trapezodial; MEDIUM; 19; 0; 36; 1; 50; 1; 65; 0
RightTrapezodial; LARGE; 50; 0; 65; 1; 101; 1

A.3 Les restructurations sous forme textuelle

TRANS#creates an abstract class from a set of sibling classes

#CLASS#factorized classes#
% DIT + [1,1]
% NMA – [0, NumMA]
% NMO + [0, NumMA]
% NMI + [NumMC, NumMC]
% NAM + [NMI, NMI]
% NAA + [NAI, NAI]
% NPM + [1/NMA, 1/NMA]
% OCMIC + [0, NumPA]

CLASS#initial super class#
% NOC – [NumFCL–1, NumFCL–1]

CLASS#classes used as type in the signature of the abstracted methods#
% OCMEC + [0, UNKNOWN]

CLASS#classes used as type in the signature of the methods created from code segments common between the factorized classes#
% OCMEC + [0, UNKNOWN]

CLASS#classes used as type for the arguments added to the abstracted methods#
% OCMEC + [0, NumPA]

CLASS#the new super class#
% DIT + [1, 1]
% NOC + [NumFCL, NumFCL]
% NMI + [0, NumMC]
% NMO + [0, NumMC]
% NMA + [0, NumMC]
% OCMIC + [0, UNKNOWN]

ENDREST

TRANS##creates subclasses#

CLASS#class to be specialized#
% NOC + [NumSCL, NumSCL]
% NMA + [NumExp, NumExp]
% OCMIC + [0, UNKNOWN]
% NAM + [0, NumExp]
% OAM – [OAM/ (NMA+NumExp), OAM/ (NMA+NumExp)]

CLASS#classes used as type for the arguments of the new methods#
% OCMEC + [0, UNKNOWN]

ENDREST

TRANS#creates of new component class #

CLASS#class that will contain the component class#
% DAC + [1, 1]
% DAC' + [1, 1]
% OCAIC + [1, 1]
% OCMIC – [0, UNKNOWN]
% NAA + [1, NumMATR]
% DAM + [NPA/ (NTA) x (NTA+1), NPA/ (NTA) x (NTA+1)]

CLASS#classes used as type for the attributes moved to the component class#
% OCAEC + [0, NumMATR]

CLASS#classes used in the signature of the methods moved to the component class#
% OCMEC – [0, UNKNOWN]

ENDREST

TRANS#converting an association modeled using inheritance into an aggregation#

CLASS#initial super class#
% NMA + [2xNAI, 2xNAI]
% NAM + [2xNAI, 2xNAI]
% DAM – [NAI/NTA, NAI/NTA]
% OAM – [2xOAMxNAI/ (NMA+2xNAI), 2xOAMxNAI/ (NMA+2xNAI)]
% NPA – [NAI, NAI]
% NOC – [1, 1]

CLASS#initial subclass#
% DIT + [1, 1]
% DAC + [1, 1]
% DAC' + [1, 1]
% NAA + [NAI+1, NAI+1]
% DAM + [NPA/ (NTA) x (NTA+1), NPA/ (NTA) x (NTA+1)]
% NMI + [0, NMA]
% NMA + [NMI–NumCM, NMI–NumCM]
% NAM + [NMI, NMI]
% NMO + [NumCM, NumCM]
% OCMIC + [0, UNKNOWN]
% OCMEC + [0, UNKNOWN]
% OCAIC + [0, NMA]

CLASS#initial super class descendant#
% NMO + [NumIATR, NumIATR]

CLASS#initial subclass descendants#
% NMI + [2xNumIATR, 2xNumIATR]

% NAM + [2xNumIATR, 2xNumIATR]

CLASS#initial subclass ancestors#
% OCAEC + [1, 1]
% OCMEC + [1, 1]

CLASS#new super class#
% NOC + [1, 1]
% OCAEC – [1, 1]

CLASS#new super class ancestors#
% OCAEC – [1, 1]
% OCMEC – [1, 1]

ENDREST