UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VERS UN SYSTÈME D'AIDE À LA DÉCISION POUR LA CONCEPTION EN GÉNIE LOGICIEL: UNE APPROCHE BASÉE SUR LES CONNAISSANCES

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

RANA BOUSLAMA

OCTOBRE 2012

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.01-2006). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

REMERCIEMENTS

J'adresse tout d'abord mes remerciements les plus sincères à mon directeur de recherche, Monsieur Hakim Lounis, professeur à l'Université du Québec à Montréal, qui m'a fait confiance en me proposant ce travail. Je tiens à luis exprimer ma profonde reconnaissance pour sa patience, sa lecture minutieuse de chacun des chapitres de ce mémoire ainsi pour ses judicieux conseils. Je le remercie pour son encadrement scientifique et humain et aussi pour son soutien financier. Ce fut un grand plaisir de travailler avec lui.

Mes remerciements vont également vers les membres de jury chargés d'évaluer ce mémoire et à tous mes enseignants de maîtrise, en particulier M. Hakim Lounis, M. Guy Tremblay, M. Ghislain Lévesque et M. Roger Villemaire.

Une grande partie de mon travail de recherche s'est faite au sein du laboratoire de recherche LATECE où l'ambiance a toujours été sympathique. Je voudrais remercier mes amis et collègues du LATECE, en particulier Oussama, Tamer, Houda, Ghizlaine et Ramdhane.

TABLE DES MATIÈRES

REMERCIEMENTS III
LISTE DES FIGURES IX
LISTE DES TABLEAUX XI
RÉSUMÉ XIII
INTRODUCTION 1
CHAPITRE I
QUALITÉ DU PRODUIT LOGICIEL 5
1.1 Introduction 5
1.2 Qualité du logiciel 5
1.2.1 Définition 5
1.2.2 Modèles de qualité 6
1.2.2.1 Le modèle de qualité de McCall 7
1.2.2.2 Facteurs liés à l'opération du produit logiciel 9
1.2.2.3 Facteurs liés à la révision du produit 9
1.2.2.4 Facteurs liés à la transition du produit 10
1.2.3 Les attributs de la qualité 10
1.2.3.1 Attributs internes de qualité 11
1.2.3.2 Attributs externes de qualité 12
1.3 Mesure de la qualité 14
1.4 Les métriques 15
1.4.1 Définition 16
1.4.2 Différents types de métriques 17
1.4.3 Exemples de métriques 17

1.4.3.1 Mesures d'héritage 20
1.4.3.2 Mesures de cohésion 23
1.4.3.3 Mesures de couplage 26
1.4.3.4 Mesures de taille et de complexité 32
1.5 Prédiction de la qualité de logiciel 34
1.6 Conclusion 34
CHAPITRE II
GESTION DES CONNAISSANCES EN GÉNIE LOGICIEL 35
2.1 Introduction 35
2.2 La connaissance 35
2.2.1 Définition 35
2.2.2 Classification des connaissances 36
2.2.3 La création de la connaissance 36
2.3 La gestion des connaissances 38
2.3.1 Le rôle de la gestion des connaissances en génie logiciel 38
2.3.2 Comment gérer le capital de connaissances 39
2.3.3 Réutilisation des connaissances en génie logiciel 41
2.3.4 Le rôle de la gestion des connaissances dans un système d'aide à la décision ... 41
2.4 Conclusion 42
CHAPITRE III
UNE APPROCHE POUR L'ACQUISITION DES CONNAISSANCES EN GÉNIE LOGICIEL 43
3.1 Introduction 43
3.2 L'expertise humaine 44
3.3 L'apprentissage automatique 44
3.3.1 Définition 45
3.3.2 Apprentissage supervisé et non supervisé 46
3.3.3 Les différentes techniques de l'apprentissage supervisé 47
3.3.4 Apprentissage d'arbres de décision 47
3.3.5 Apprentissage automatique de règles 49
3.3.5.1 Règles propositionnelles et règles en logique du premier ordre ... 50

3.3.5.2 Algorithmes de couverture 52

3.3.5.1 Biais utilisés pour l'apprentissage de règles 54

3.3.6 Quelques systèmes d'apprentissage de règles 58

3.3.6.1 Le système AQ 58

3.3.6.2 Le système CN2 58

3.3.6.3 Le système FOIL 59

3.4 Conclusion 60

CHAPITRE IV
ENVIRONNEMENT EXPERIMENTAL 61

4.1 Introduction 61

4.2 Description de Weka 61

4.2.1 Apprentissage avec Weka 62

4.2.2 Préparation de données d'apprentissage (Preprocess) 63

4.2.3 Application d'un algorithme d'apprentissage 65

4.2.4 Les algorithmes utilisés dans notre approche 66

4.2.4.1 J48 66

4.2.4.2 Part 68

4.2.4.3 ConjuntiveRule 69

4.2.4.4 JRIP 70

4.3 Description de JRules 71

4.3.1 Le moteur d'inférence d'ILOG JRules 71

4.3.2 Règles JRules 73

4.3.3 Agenda 74

4.3.4 Algorithme RETE 75

4.4 Conclusion 76

CHAPITRE V
INPLÉMENTATION DU PROTOTYPE ET EXPÉRIMENTATION 77

5.1 Introduction 77

5.2 Modules du prototype 77

5.3 Éclipse et Java comme infrastructure d'implémentation 78

5.3.1 Java 78

5.3.2 Éclipse 78
5.4 Diagramme de cas d'utilisation 79
5.4.1 Les acteurs 80
5.4.2 Les différents cas d'utilisation 80
5.5 Expérimentation 81
5.5.1 Création d'un fichier arff à partir d'un fichier Excel 83
5.5.2 Création d'une classe Java qui correspond à l'hypothèse Hypothèse-COMP_REUT 85
5.5.3 Apprentissage automatique 87
5.5.4 Construction d'une base de règle (fichier .ilr) 90
5.5.5 Consultation ou modification d'une base de règles 92
5.5.6 La prédiction 95
5.5.6.1 Description de l'interface de prédiction 98
5.5.6.2 Analyse des résultats de prédiction 100
5.6 Conclusion 104
CONCLUSION 105
BIBLIOGRAPHIE 109

LISTE DES FIGURES

Figure		Page
0.1	Système d'aide à la décision basée sur les connaissances, (*Lounis et al.*, 2009)	2
1.1	Modèle de qualité de McCall (McCall et al. 1977)................................	8
1.2	Processus de mesure des métriques..	16
2.1	Capitalisation des connaissances (Manuel Zacklad et Mickel Grundstein).......	40
4.1	L'interface Explorer de Weka..	63
4.2	Exemple d'un fichier arff...	64
4.3	Résultats d'apprentissage..	66
4.4	Résultats de classification de l'algorithme J48..................................	67
4.5	Résultats d'exécution de l'algorithme Part..	68
4.6	Résultats d'exécution de l'algorithme ConjuntiveRule............................	69
4.7	Résultats d'exécution de l'algorithme JRIP..	70
4.8	Moteur d'inférence de JRules..	72
4.9	Réseau RETE..	76
5.1	Diagramme de cas d'utilisation de système d'aide à la décision..................	79
5.2	Extrait d'un fichier Excel d'une base de données.................................	84
5.3	Résultat de conversion du fichier Excel en un fichier 'arff'........................	85
5.4	Création d'une classe Java des objets à prédire..................................	87
5.5	Résultat d'apprentissage avec l'algorithme Part..................................	90
5.6	Exemple de règles JRules..	91
5.7	Création d'un fichier .ilr...	92
5.8	Consultation d'une base de connaissance par un débutant........................	93
5.9	Consultation et/ou modification d'une base de connaissances par un expert.....	94
5.10	Liste d'objets à prédire dans un fichier Excel....................................	96

5.11 Exemple d'un ensemble d'objets à insérer dans la mémoire de travail de JRules 97
5.12 Exemple d'exécution d'un modèle pour prédire la réutilisabilité de composants logiciels.. 99

LISTE DES TABLEAUX

Tableau Page

2.1 Modèle de partage de connaissances (*Nonaka*, 1994)........................... 37

5.1 Résultats de classification des composants logiciels............................ 102

RÉSUMÉ

Les métriques logicielles jouent un rôle très important dans la prédiction de la qualité. Elles aident les gestionnaires dans la prise de décisions afin de budgétiser, contrôler, estimer le coût et analyser les risques d'un produit au cours de son développement.

Dans ce travail, nous proposons une approche basée sur les connaissances pour analyser et estimer des facteurs de qualité dans des systèmes à objets. Pour concrétiser notre approche, nous avons construit un prototype regroupant les fonctionnalités de deux logiciels. Nous avons utilisé le logiciel Weka pour faire l'apprentissage automatique de connaissances et ainsi construire des modèles prédictifs. Ensuite, nous avons traduit ces modèles en un système à base de règle JRules, pour la prédiction et la prise de décision. Ces deux fonctions principales sont offertes pour deux types d'utilisateur : un débutant et un expert dans le domaine de la conception en génie logiciel. Le rôle principal de l'expert est de valider un tel modèle prédictif.

Nous avons expérimenté notre prototype sur des bases de données qui représentent des mesures de métriques récoltées sur des logiciels fonctionnels. Les résultats obtenus dans le cadre de différentes expériences permettent de prédire et d'estimer certains facteurs de qualité tels que la maintenabilité, la réutilisabilité et la prédisposition aux fautes.

Mots Clés :

Qualité du logiciel, apprentissage automatique, base de connaissances, modèles prédictifs, système à base de règles, prise de décision.

INTRODUCTION

Les organisations ont un intérêt certain pour la technologie orientée objet, qui offre une autre vision du développement du produit logiciel et promet un gain de temps aussi bien en conception et codage qu'en suivi et maintenance.

Dans de nombreux systèmes orientés objet, des défauts de conception introduits lors des premières étapes de la conception ou au cours de l'évolution des systèmes, sont des causes fréquentes de faibles maintenabilité, haute complexité et de comportement erroné des programmes. La préservation d'une conception correcte doit être une préoccupation constante. Toutefois, dans les systèmes qui ont un grand nombre de classes et qui sont sujet à de fréquentes modifications, la détection et la correction des défauts de conception peuvent être une tâche complexe.

Depuis plusieurs années, les organisations du génie logiciel s'intéressent de plus en plus à la qualité, à ses caractéristiques, à ses indicateurs et aux modèles qui permettent de la prédire. Habituellement, le but de ces modèles est de prédire un facteur de qualité à partir d'un ensemble de mesures directes. Plusieurs travaux sur l'estimation de la qualité proposent des solutions permettant de détecter des anomalies à l'aide de métriques, de profiter de ce qui a déjà été fait pour mieux faire lors des réalisations futures en établissant certaines règles et modèles d'estimation pour améliorer la qualité.

Les gestionnaires de développement du logiciel doivent donc prendre des décisions importantes pour prédire la qualité des logiciels. La prise de décision n'est pas une tâche facile ; elle est d'une difficulté inhérente qui est aggravée par la complexité et la rapidité des changements qui caractérisent le génie logiciel. Des décisions critiques doivent être basées sur des connaissances.

De nos jours, deux types de méthodes sont particulièrement utilisées pour acquérir ces connaissances et pour construire des modèles prédictifs : les méthodes par expertise humaine et les méthodes par apprentissage automatique. La première approche se présente sous forme d'heuristique, elle permet de capitaliser l'expérience des personnes expertes dans un domaine particulier et de mémoriser leurs connaissances sur un support physique (papier, disque, ..). Tandis que les méthodes par apprentissage automatique sont présentées comme des systèmes intelligents de support à la décision. Ces systèmes supportent la prise de décision en coordonnant certaines activités telles que la cueillette des données, l'extraction des données, la génération de concepts, l'induction des règles et des modèles, l'interaction avec les usagers et enfin la suggestion de solutions.

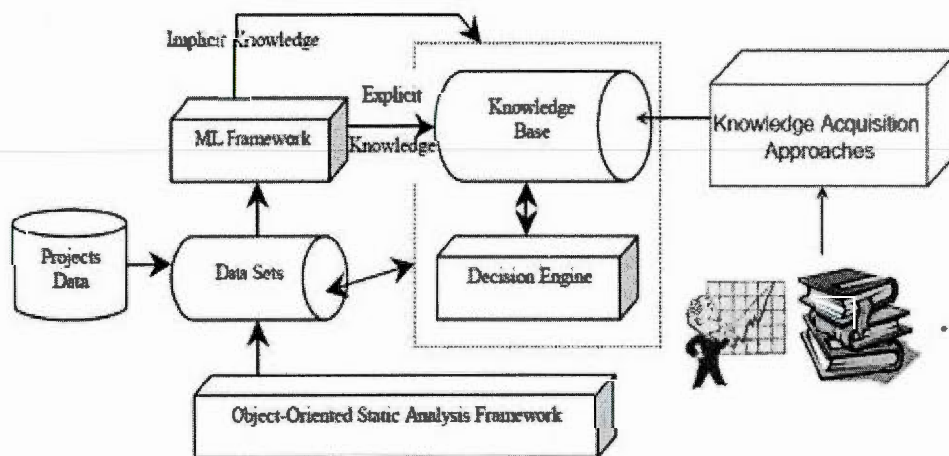

Figure 0.1 Système d'aide à la décision basée sur les connaissances, (*Lounis et al.*, 2009)

La première de ces activités consiste à recueillir des données provenant de statistiques ou d'observations dans des bases de données (BD). Les méthodes d'apprentissage automatique interviennent alors pour générer des concepts à partir de la BD sur la base de certaines heuristiques et en induire des règles et des modèles. Ces règles peuvent être conservées dans des bases de connaissances ou extraites directement par un utilisateur via une interface dans le but de supporter la prise d'une décision.

Notre étude rentre dans ce cadre et le travail que nous présentons porte sur le développement d'un prototype dont l'objectif est la prédiction de la qualité et prise de la bonne décision pour la conception. Il s'agit de générer des règles par apprentissage automatique à partir d'un ensemble de données, ensuite d'affiner ces règles par un expert du domaine et enfin les utiliser sur de nouvelles données au sein d'un système à base de règles.

Le mémoire comporte cinq chapitres. Le premier chapitre vise à introduire le génie logiciel et à définir ses concepts : logiciel, génie logiciel, qualité, mesure et attributs de qualité internes et externes ainsi que les métriques.

Le deuxième chapitre est une description des connaissances en génie logiciel et une présentation de différentes méthodes pour les capitaliser.

Le troisième chapitre traite de deux approches différentes pour gérer les connaissances : l'expertise humaine et l'apprentissage automatique.

Le quatrième chapitre est consacré à notre approche. Il comprend une description des algorithmes sélectionnés pour faire l'apprentissage et la génération de règles. On y décrit le système à base de règles dont le but est la prise de décision lors de futures conceptions.

Finalement, Le dernier chapitre décrit notre prototype qui aide à la prédiction de la qualité du logiciel ainsi que les résultats obtenus durant les expérimentations.

Une conclusion qui fait le point sur les contributions de ce travail et met en avant les perspectives envisagées.

CHAPITRE I

QUALITÉ DU PRODUIT LOGICIEL

1.1 Introduction

Le but principal du développement de logiciels est de produire des logiciels de bonne qualité. Le terme « qualité » est un terme qui nous est très familier ; il touche plusieurs aspects du logiciel. En général, c'est le fait de développer un logiciel qui satisfait tous les besoins de son utilisateur. Les premiers travaux liés à la mesure de la qualité des logiciels ont été évoqués par (McCall *et al.* 1977) et (Boehm *et al.*, 1978).

Afin d'approfondir nos connaissances sur la qualité d'un produit logiciel, ce chapitre donne un aperçu sur les différents aspects du logiciel. Nous commençons par la définition d'un logiciel, les facteurs de qualité et les attributs de qualité. Ensuite, nous avons traité les différentes techniques utilisées pour améliorer la qualité d'un logiciel et nous présentons un axe important, celui de la mesure.

1.2 Qualité du logiciel

1.2.1 Définition

En génie logiciel, la qualité logiciel est une vue globale du logiciel, basée sur de nombreux critères : la précision des résultats, la tolérance aux pannes, la simplicité, l'extensibilité, la compatibilité, la portabilité, la facilité de correction et de transformation, la performance, la consistance et l'intégrité des informations.

Les problèmes de qualité des logiciels sont connus depuis les années 1960 et ils sont toujours liés au respect des coûts, des délais, du cahier des charges et du niveau de la qualité. Nous trouvons dans la littérature de multiples définitions de la qualité. En voici quelques-unes :

- Kitchenham (1996) mentionne que « La qualité est difficile à définir, impossible de la mesurer, mais facile à connaître »

- Selon Gillies (1992), la qualité est « transparente quand elle est présente, mais facilement reconnue en son absence »

- La norme ISO 9126 (ISO/IEC 9126, 1991), définit six groupes d'indicateurs de qualité des logiciels : la capacité fonctionnelle, la facilité d'utilisation, la fiabilité, la performance, la maintenabilité et la portabilité. C'est donc la conformité à des besoins fonctionnels et à des caractéristiques implicites attendues de tout logiciel développé.

- Bell et al. (1992) définit la qualité par rapport à l'utilisateur final et au logiciel livré. Il définit un logiciel de qualité comme étant celui qui satisfait les besoins des utilisateurs, est fiable et facile à maintenir.

1.2.2 Modèles de qualité

On trouve dans la littérature plusieurs modèles de qualité du logiciel. James McCall a rédigé en 1977 un des premiers documents de discussion de l'industrie du logiciel sur les facteurs de qualité du logiciel. Son modèle, nommé « modèle McCall », vise surtout les développeurs et il est utilisé durant le processus de réalisation. Il a atteint au cours de son évolution un certain niveau de maturité en fournissant une méthode pratique pour classifier les spécifications logicielles.

1.2.2.1 Le modèle de qualité de McCall

Jim McCall est l'un des premiers qualiticiens du logiciel qui a essayé de quantifier la qualité du produit logiciel.

«In his quality model, McCall attempts to bridge the gap between users and developers by focusing on a number of software quality factors that reflect both the users views and the developers priorities » (Midilic D, 2005).

« McCall, dans son modèle, a essayé d'établir le lien entre les utilisateurs et les développeurs, en se concentrant sur un certain nombre de facteurs de qualité du logiciel qui reflètent les vues des utilisateurs et les priorités des réalisateurs».

Pour construire son modèle, il a proposé une décomposition et une structuration du concept de « qualité ». Ce modèle est structuré sur quatre niveaux (figure 1.1) :

- Le premier niveau divise le cycle de vie du produit logiciel en trois classes :
  - Opération : l'utilisation du produit ;
  - Révision : changement pour l'entretien ;
  - Transition : changement pour s'adapter à un nouvel environnement de travail.
- Le deuxième niveau représente les « facteurs de qualité ». Ces facteurs décrivent la perception de l'utilisateur et du client (point de vue externe). McCall a défini 11 principaux facteurs à utiliser pour analyser la qualité du logiciel. Ces facteurs sont les grands axes à vérifier pour savoir si un logiciel est de bonne ou de mauvaise qualité. Ces 11 facteurs sont groupés en trois catégories :
  - Facteurs liés au fonctionnement du produit logiciel : exactitude, fiabilité, efficacité, intégrité et utilisabilité.
  - Facteurs de révision du produit : maintenabilité, flexibilité et testabilité.

> ➢ Facteurs de transition du produit : portabilité, réutilisabilité et interopérabilité.

- Le troisième niveau est constitué des « critères de qualité ». Les critères sont des propriétés internes au logiciel et ils concernent les développeurs. Ils sont au nombre de 23.

- Le dernier niveau concerne les métriques qui s'appliquent sur les propriétés internes et qui sont mesurables.

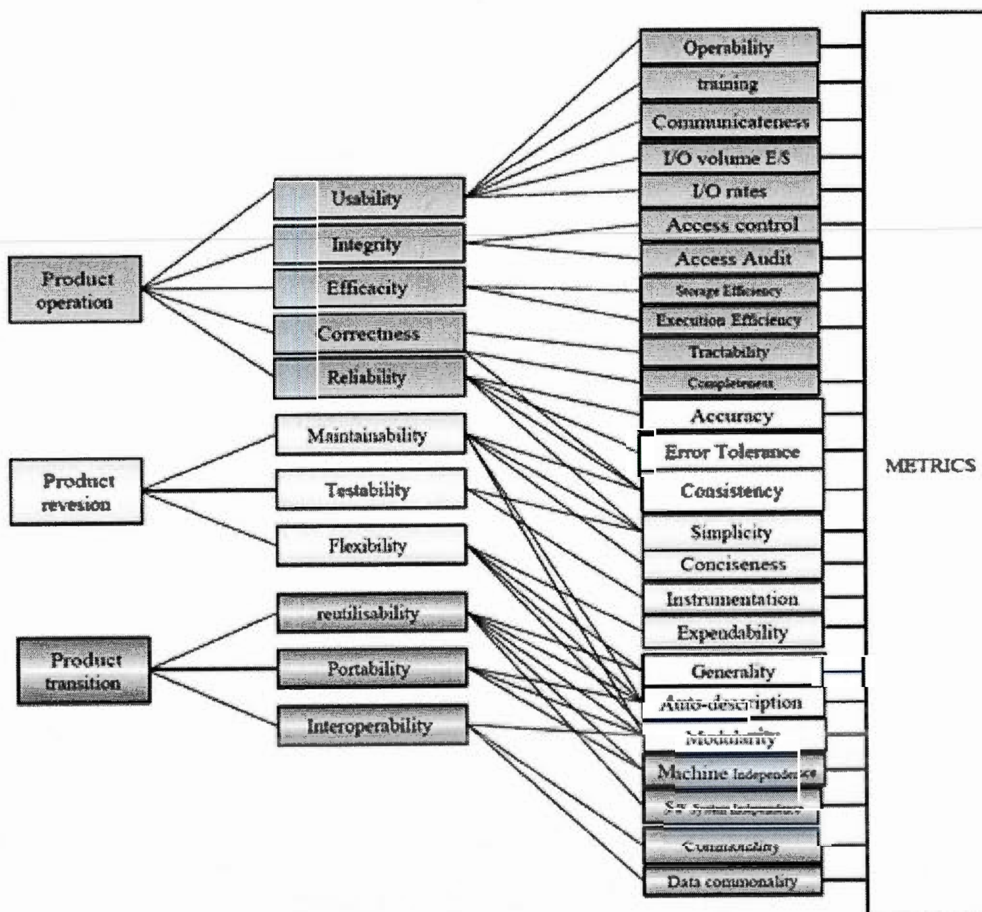

Figure 1.1 Modèle de qualité de McCall (McCall et al. 1977)

1.2.2.2 Facteurs liés à l'opération du produit logiciel

- Exactitude : c'est l'aptitude d'un programme ou d'un produit logiciel à remplir conformément ses fonctions qui sont définies dans le cahier des charges par le client.

- Fiabilité : c'est l'aptitude d'un produit logiciel à fonctionner dans des conditions anormales sans pannes, pour une durée d'utilisation donnée.

- Efficacité : ce facteur mesure l'aptitude d'un logiciel à optimiser la consommation des ressources matérielles qu'il utilise (CPU, E/S, mémoire, etc.).

- Intégrité : c'est l'aptitude du logiciel à protéger les données qu'il manipule contre les accès non autorisés et malveillants.

- Utilisabilité : elle concerne la facilité d'utilisation, d'apprentissage d'un nouvel utilisateur, de préparation des données ou de fonctionnement d'un système logiciel.

1.2.2.3 Facteurs liés à la révision du produit

- Maintenabilité : ces exigences déterminent l'aptitude du logiciel à faciliter la localisation et la correction rapide d'erreurs résiduelles quand le système est en phase d'exploitation, puis à vérifier la fiabilité de ces corrections. Il n'existe pas de métriques qui mesurent directement la maintenabilité, mais le coût de la correction peut nous donner une idée sur ce facteur.

- Flexibilité : les ressources et les efforts exigés pour supporter les activités de maintenance adaptative sont couverts par les exigences de flexibilité. Ceci inclut les ressources nécessaires pour adapter un package logiciel à un ensemble de clients.

- Testabilité : elle mesure l'aptitude du logiciel à être testé et vérifié afin de s'assurer que tous ses composants travaillent dans l'ordre spécifié. Il faut noter que le système est en phase d'exploitation.

1.2.2.4 Facteurs liés à la transition du produit

- Portabilité : c'est la facilité avec laquelle un logiciel peut être transféré vers d'autres environnements composés de différents matériels et de différents systèmes d'exploitation.

- Réutilisabilité : c'est l'aptitude d'un logiciel à être réutilisé, en totalité ou en partie, dans un nouveau projet logiciel en cours de développement.

- Interopérabilité : elle mesure l'aptitude d'un logiciel à garantir l'ouverture du système à d'autre système.

1.2.3 Les attributs de la qualité

Une qualité importante du logiciel bien conçu est d'être modulaire. Deux concepts qui permettent d'évaluer dans quelle mesure le logiciel est modulaire sont la cohésion et le couplage (*Lounis et al.,* 1998).

Un module est défini comme une séquence d'instructions contiguës identifiées par un nom. Un produit logiciel est dit modulaire si ses composants présentent un haut degré de cohésion et un faible degré de couplage.

La modularité est un concept très important dans la conception d'un produit logiciel selon la méthodologie orienté objet. Il est impératif donc que les modules soient bien conçus pour assurer un certain niveau de qualités puisqu'ils peuvent influer sur différentes caractéristiques de qualité comme l'efficacité, la flexibilité, la facilité de maintenance et la facilité de réutilisation.

Nous souhaitons que nos systèmes logiciels soient rapides, fiables, faciles à utiliser, simples à lire, modulaires, structurés, etc. Mais ces adjectifs recouvrent deux classes de qualités bien différentes. D'un côté, nous envisageons des qualités comme la vitesse ou la facilité d'utilisation, dont la présence, ou non, dans un produit logiciel peut être détectée par ses utilisateurs. Ces propriétés peuvent être appelées des facteurs externes de qualité. Les

autres qualités d'un produit logiciel, comme celles d'être modulaire ou facile à lire, sont des facteurs internes, seulement perceptibles aux informaticiens professionnels qui ont accès au code source du logiciel.

1.2.3.1 Attributs internes de qualité

Les attributs internes de qualité les plus importants dans le développement des logiciels orientés objet sont : l'héritage, le couplage, la cohésion, la taille et la complexité.

- Couplage

Le couplage entre deux modules décrit la force des associations qui existent entre deux modules d'un système. Il dépend de la complexité des interfaces entre modules. Intuitivement un faible degré de couplage entre les modules est souhaitable car en effet, un module relativement peu relié aux autres est facile à comprendre, à modifier, à tester et enfin à réutiliser. De plus, les chances de propagation des erreurs d'un module à un autre se trouvent d'autant plus diminuées que le nombre de liens entre ces deux modules est petit.

- Cohésion

De même que le couplage, la notion de cohésion est à la base des bonnes pratiques en conception orientée objet (Philippe, 2005). Elle mesure la force de dépendance entre les éléments composant un module donné. La cohésion d'une classe représente le fait que ses responsabilités forment un tout cohérent.

Un module possède un haut degré de cohésion si tous ces éléments sont reliés entre eux. Ces éléments collaborent pour réaliser un objectif commun qui est la fonction du module. Ainsi, il est important d'assurer une cohésion élevée pour avoir un certain niveau de qualité.

- Héritage

L'héritage est une caractéristique très importante pour la conception orientée objet. C'est un mécanisme qui permet de propager les fonctionnalités d'un objet à d'autres. Il se traduit par une hiérarchie de classes de plusieurs niveaux.

1.2.3.2 Attributs externes de qualité

Les attributs externes, que nous traitons dans cette section sont la maintenabilité, la réutilisabilité et la prédisposition aux fautes.

- Maintenabilité

La maintenance du logiciel est définie dans la norme IEEE 1219, comme la facilité selon laquelle un logiciel après la livraison peut être maintenu, amélioré au niveau des performances ou adapté à un nouvel environnement. La maintenance consomme plus de 60% des ressources totales investies dans le développement du produit logiciel à travers le cycle de vie (Pressman, R.S, 1997). Au niveau de la littérature, nous retrouvons différents types de maintenance.

Les catégories de maintenance définies par ISO/IEC sont les suivantes :

✓ Maintenance corrective : modification réactive d'un produit logiciel effectuée après la livraison au client, due à la non-conformité aux fonctionnalités décrites dans la documentation d'utilisation du logiciel. Ce type de maintenance permet alors de corriger les erreurs trouvées dans le code et danse la documentation.

✓ Maintenance adaptative : elle désigne les modifications à apporter sur un produit logiciel après la livraison en tenant compte des changements et de l'évolution de l'environnement logiciel et matériel, et aussi des nouvelles exigences du client, sans modifier le produit logiciel de base.

✓ La maintenance d'amélioration des fonctionnalités : elle rassemble deux sous-types de maintenance : la maintenance perfective (modification d'un produit logiciel après sa livraison pour étendre la performance) et la maintenance préventive (modification d'un produit logiciel après sa livraison pour détecter ou corriger des défauts latents dans le produit logiciel avant qu'ils ne deviennent des défaillances).

- Réutilisabilité

La réutilisabilité est souvent définie comme : « l'aptitude du logiciel à être réutilisé, en entier ou en partie pour de nouvelles applications » (Bailly, 1987)

La réutilisabilité est une caractéristique très importante des composants logiciels de haute qualité. Elle permet d'accroitre la productivité en réduisant le temps de développement. Elle nous garantit aussi la fiabilité, puisqu' il s'agit d'utiliser à nouveau un composant fonctionnel.

C'est pour cette raison, qu'un composant logiciel doit être conçu et implémenté de telle manière qu'il puisse être réutilisé dans d'autres systèmes logiciels. Il existe plusieurs entités que l'on peut réutiliser. Selon Scott Amber, les niveaux de réutilisation sont les suivantes (Scott Ambler, 1998) :

- ✓ La réutilisation du code;
- ✓ La réutilisation avec l'héritage;
- ✓ La réutilisation de documentation;
- ✓ La réutilisation des composants;
- ✓ La réutilisation des artéfacts;
- ✓ La réutilisation des patterns;
- ✓ La réutilisation des composants de domaine.

- Prédisposition aux fautes

Un des objectifs d'une organisation de développement de logiciel est la réduction du risque d'avoir des défauts dans les composants logiciels afin d'améliorer la productivité. Un produit sans erreurs réduira l'effort de correction et de maintenance.

1.3 Mesure de la qualité

Vers la fin du 19ème siècle, le physicien Lord Kelvin disait de la mesure: « *When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science*». (*Pressman et Roger S*, 2004)

La mesure est un processus qui consiste à mettre en correspondance des choses appartenant à un monde empirique *'what you are speaking about'* et d'autres choses appartenant au monde mathématique définit par des nombres et des symboles. Les scientifiques ont toujours fait des efforts pour rendre les choses mesurables afin d'offrir un moyen rigoureux de manipulation et d'expérimentation du monde qui nous entoure.

Les mesures sont importantes non seulement dans le domaine des sciences exactes (exemple, physique, chimie, etc.), mais aussi en génie logiciel. Cela nous aide à mieux comprendre et à maîtriser le processus de développement des logiciels.

Avec l'évolution du génie logiciel et de ses techniques, plusieurs mesures ont été proposées dans la littérature afin de répondre aux différents besoins en matière de qualité.

D'un autre côté, il y a les mesures qui sont conçues dans le cadre du paradigme structuré ou dans le cadre de l'orienté-objet (*Abreu et carapuça*, 1994); (*Briand, Daly et Wuest*, 1997); (*Briand, Devanbu et Melo*, 1997); (*Chidamber et Kemerer*, 1991); (*Li et Henry*, 1993) ; (*Lorenz et Kidd*, 1994) ; Ce sont des mesures qui touchent les attributs internes de la conception du logiciel, dont le couplage, l'héritage, la cohésion, la taille, etc.

D'un autre côté, il existe d'autres mesures qui sont les mesures proposés par ISO 9126. Par exemple ISO TR 9126-2 (2003a) et ISO TR 9126-3 (2003b) proposent un ensemble de mesures qui permettent de mesurer d'une façon quantitative les caractéristiques de qualité interne et externe du produit logiciel, dont la capacité fonctionnelle (tel que

l'aptitude, l'exactitude, etc.), la fiabilité (maturité, tolérance aux fautes, etc.), la facilité d'utilisation (facilité de compréhension, facilité d'apprentissage, facilité d'exploitation, etc.), le rendement (comportement vis-à-vis du temps, ressources utilisées, etc.), la maintenabilité (facilité d'analyse, facilité de modification, stabilité, etc.) et la portabilité (facilité d'adaptation, facilité d'installation, etc.). ISO TR 9126-4 (2004) propose un ensemble de mesures pour les caractéristiques de qualité en mode utilisation du produit logiciel, dont l'efficacité, la productivité, la sécurité et la satisfaction.

Pour améliorer et faire évoluer le domaine de la mesure, il faut savoir d'abord l'utilité de la mesure pour le logiciel : « pourquoi a-t-on besoin de mesurer le logiciel ? ». En effet, on ne mesure pas un logiciel pour le plaisir de le mesurer, mais pour des objectifs bien définis. Oman et Pfleeger (1997) ont distingué six objectifs pour mesurer :

- mesurer pour comprendre le produit logiciel
- mesurer pour l'expérimentation
- mesurer pour faciliter le contrôle de la gestion de projet
- mesurer pour l'amélioration du processus.
- mesurer pour identifier les situations qui nécessitent l'amélioration du produit logiciel au cours de la maintenance ou bien de la réutilisation.
- et mesurer pour supporter les gestionnaires dans leurs prises de décision et pour la prédiction.

1.4 Les métriques

La qualité du produit logiciel est le critère majeur pour son acceptation et son succès. Le problème de base en gestion de projets est de fournir un produit logiciel en respectant le budget, les échéances et un certain degré de qualité.

Une bonne compréhension et maîtrise de la complexité d'un code source du logiciel permet de développer un logiciel de meilleure qualité. Pour garantir un logiciel de qualité tout

en assurant des coûts de test et de maintenance faibles, la complexité du logiciel devrait être mesurée. Pour quantifier cette complexité, on se sert de métriques.

Les métriques sont issues de la littérature, d'outils existants, de guides méthodologiques, ou de la connaissance empirique des experts du domaine.

1.4.1 Définition

Une métrique est une échelle quantitative et une méthode qui peuvent être employées pour fournir une indication sur la complexité de la conception et du code source d'un produit logiciel spécifique. Elle peut aussi servir à faire des tests efficaces au cours de la conception. Avant d'être utilisée, une métrique passe par un processus de mesure tel qu'illustré dans les travaux de Walnut Creek en 1997.

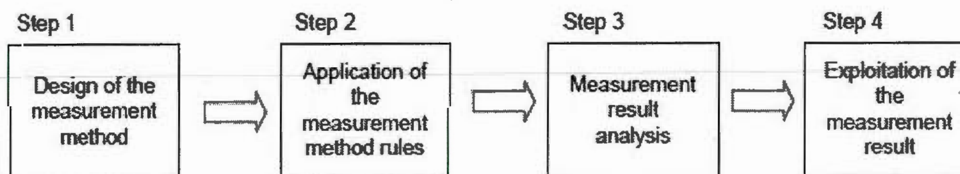

Figure 1.2 Processus de mesure des métriques

1ère étape : elle consiste à construire une méthode de mesure avant de commencer le processus de mesure.

2ème étape : il faut appliquer les règles de la méthode de mesure sur la totalité de logiciel ou bien sur une partie.

3ème étape : obtenir et analyser les résultats de la mesure.

4ème étape : exploitation des résultats dans des modèles quantitatifs ou qualitatifs.

1.4.2 Différents types de métriques

Afin de mesurer la qualité du logiciel, des métriques sont utilisées dans plusieurs travaux de recherche et sont associées aux différents facteurs de qualité. Les métriques peuvent être classées sur différentes catégories :

- métriques de processus : elles décrivent des propriétés relatives au cycle de développement comme l'effort de développement et la productivité.
- métrique du produit logiciel : elles décrivent des propriétés du logiciel comme : la taille, complexité cyclomatique, nombre de fonctions, etc.
- les métriques de la spécification et conception : elles décrivent la fréquence de communication de données entres les modules, le couplage, la cohésion, etc.
- les métriques mesurant la taille et la complexité du code : comme les métriques de lignes de code, Taux des commentaires, nombres d'opérandes et d'opérateurs, etc.

Dans le cadre de ce travail, nous traiterons les métriques de produits, de code et de spécification.

1.4.3 Exemples de métriques

L'objet de cette section est de présenter des mesures orientées objets qui seront utilisées pour réaliser nos expérimentations. Nous avons utilisé la terminologie utilisée dans les travaux de Mao Y. en 1998.

<u>Relations d'héritage dans les systèmes orientés objet :</u>

Soit un système orienté objet composé d'un ensemble de classes noté C, on peut définir les relations d'héritage pour chaque classe $c \in C$ comme les suivants :

$Parents(c) \subset C$: l'ensemble des classes parents de la classe c

$Children(c) \subset C$: l'ensemble des classes enfants de la classe c

$Ancestors(c) \subset C$: l'ensemble des classes ancêtres de la classe c

$Descendants(c) \subset C$: l'ensemble de classes descendantes de c

Une classe c peut considérer une classe d comme ami si d peut utiliser les éléments non publics de c, d'où :

$Friends(c) \subset C$: l'ensemble des classes amis de la classe c

$Friends'(c) \subset C$: l'ensemble des classes qui déclarent la classe c comme ami

$Others(c) \subset C$: l'ensemble des autres classes par rapport à la classe c

Sachant que :

$$Others(c) = C \setminus (Ancestors(c) \cup Descendants(c) \cup Friends(c) \cup Friends'(c) \cup \{c\})$$

Méthodes :

Une classe c a un ensemble de méthodes $M(c)$. Une méthode peut être virtuelle ou non virtuelle, héritée, surchargée ou nouvellement définie.

On peut deviser les méthodes de la classe c en deux parties :

$M_D(c)$: l'ensemble des méthodes déclarées de c

$M_I(c)$: l'ensemble des méthodes implémentées de c

Tel que : $M(c) = M_D(c) \cup M_I(c)$ et $M_D(c) \cap M_I(c) = \Phi$

Une autre façon de diviser l'ensemble des méthodes est la suivante :

$M_{INH}(c) \subseteq M(c)$: l'ensemble des méthodes héritées de la classe c

$M_{OVR}(c) \subseteq M(c)$: l'ensemble des méthodes surchargées de la classe c

$M_{NEW}(c) \subseteq M(c)$: l'ensemble des méthodes nouvelles, non héritées et non surchargées de la classe c,

Tel que : $M(c) = M_{INH}(c) \cup M_{OVR}(c) \cup M_{NEW}(c)$

On peut prendre en considération l'ensemble des méthodes publiques et les non publiques (privées) :

$M_{Pub}(c) \subseteq M(c)$: l'ensemble des méthodes publiques

$M_{npub}(c) \subseteq M(c)$: l'ensemble des méthodes non publiques

Tel que : $M(c) = M_{Pub}(c) \cup M_{npub}$

Attributs :

Une classe a des attributs hérités ou nouvellement déclarés. Pour chaque classe $c \in C$, soit A(c) l'ensemble des attributs de la classe c :

$A(c) = A_D(c) \cup A_I(c)$ Où

$A_D(c)$: l'ensemble des attributs déclarés dans la classe c

$A_I(c)$: l'ensemble des attributs implémentés dans la classe c

On représente les attributs référencés par les méthodes par:

$AR(m)$: l'ensemble des attributs référencés par la méthode m, tel que $m \in M(c)$

$NAR(m,a)$: le nombre de références de la méthode m à l'attribut a, tel que $a \in A(d)$ pour $d \in C$.

Prédicats :

Pour déterminer le couplage des classes $c, d \in C$, on définit le prédicat suivant:

$$uses(c,d) \Leftrightarrow (\exists m \in M_I(c) : \exists m' \in M_I(d) : \exists m' \in SIM(m))$$
$$\vee (\exists m \in M_I(c) : \exists a \in A_I(d) : \exists a \in AR(m))$$

Tel que : $SIM(m)$ est l'ensemble des méthodes statiques invoquées de *m*.

Interactions :

Pour déterminer certaines interactions entre classes, $c, d \in C$, on définit par :

ACA(c, d) : le nombre actuel des interactions classe-attribut d'un serveur *c* à un client *d*.

ACM(c, d) : le nombre actuel des interactions classe-méthode d'un serveur *c* à un client *d*.

AMM(c, d) : le nombre actuel des interactions méthode-méthode d'un serveur *c* à un client *d*.

1.4.3.1 Mesures d'héritage

Ces métriques ont été définies dans les travaux de Chidamber S.R. et Kemerer C.F en 1997. Elles mesurent la position d'une classe dans l'arbre d'héritage, le nombre de superclasses, le nombre de sous-classes et de méthodes héritées :

DIT, Depth of Inheritance Tree :

Le *DIT* d'une classe est la distance maximale entre une classe et la classe racine de l'arbre d'héritage. En cas d'héritage multiple, *DIT* vaudra la profondeur maximale depuis la classe racine jusqu'à la classe mesurée.

$$DIT(c) = \begin{cases} 0, si\ parents(c) = 0 \\ 1 + \max\{DIT(c') : c' \in Parents(c)\}, \text{sinon} \end{cases}$$

AID, Average Inheritance Depth of a class :

C'est la profondeur moyenne de l'arbre d'héritage. L'*AID* d'une classe sans des descendants est égale toujours à zéro

$$AID(c) = \begin{cases} 0 \ si \ parents(c) = 0 \\ \dfrac{\sum_{c' \in Parents(c)} (1 + AID(c'))}{|Parents(c)|}, \text{ sinon} \end{cases}$$

CLD, Class-to-Leaf Depth :

Le CLD d'une classe est le nombre maximal de niveau d'héritage de cette classe à une feuille qui appartient à la même hiérarchie.

$$CLD(c) = \begin{cases} 0, si \text{ Descendants}(c) = 0 \\ \max\{DIT(c') - DIT(c) : c' \in Descendants(c)\}, \text{ sinon} \end{cases}$$

NOC, Number Of Children :

Le nombre des sous classes qui héritent directement d'une classe donnée

$$NOC(c) = |Children(c)|$$

NOP, Number Of Parents :

C'est le nombre de classes à partir desquelles une classe donnée hérite directement.

$$NOP(c) = |Parents(c)|$$

NOD, Number Of Descendents :

C'est le nombre de classes qui héritent directement ou indirectement d'une classe.

$$NOD(c) = |Descendants(c)|$$

NOA, Number Of Ancestors :

C'est le nombre de classes distinctes à partir desquelles une classe donnée hérite directement ou indirectement. Cette mesure donne le même résultat que la métrique *DIT* si l'héritage multiple n'est pas utilisé.

$$NOA(c) = |Ancestors(c)|$$

NMO, Number of Methods Overridden :

Le nombre de méthodes surchargées.

$$NMO(c) = |M_{OVR}(c)|$$

NMI, Number of Methods Inherited :

Le nombre de méthodes héritées mais non surchargées.

$$NMI(c) = |M_{INH}(c)|$$

NMA, Number of Methods Added, new methods :

Le nombre de nouvelles méthodes dans une classe, qui ne sont ni héritée, ni surchargées.

$$NMA(c) = |M_{NEW}(c)|$$

SIX, Specialization Index :

Index de spécialisation défini par le rapport du produit *NMO* et *DIT* sur le nombre total de méthodes d'une classe donnée.

$$SIX(c) = \frac{NMO(c).DIT(c)}{|M(c)|}$$

1.4.3.2 Mesures de cohésion

De même que le couplage, la notion de cohésion est à la base des bonnes pratiques en conception orientée objet. La cohésion d'une classe représente le fait que toutes ses responsabilités forment un tout cohérent.

Un module possède un haut degré de cohésion si tous ses éléments sont reliés entre eux. Ses éléments collaborent pour réaliser un objectif commun qui est la fonction du module. Ainsi, il est important d'assurer une cohésion élevée pour avoir un certain niveau de qualité. Les principales métriques de cohésion sont :

Lack of Cohesion in Methods ($LCOM_1$) :

C'est le nombre de paires de méthodes disjointes qui n'utilisent pas des attributs en commun.

$$LCOM_1(c) = \left|\{[m_1, m_2] : m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \Phi\}\right|$$

Lack of Cohesion in Methods ($LCOM_2$) :

$LCOM_2(c)$ est le nombre de composants connectés de G_c, tel que $G_c = (V_c, E_c)$ est un graphe unidirectionnel, $V_c = M_I(c)$ et E_c les arcs définis par :

$$E_c(c) = \{[m_1, m_2] : m_1, m_2 \in V_c(c) \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \Phi\}$$

$LCOM_2$ permet d'évaluer l'encapsulation, évaluer la complexité et évaluer les défauts de conception de classes.

Lack of Cohesion in Methods ($LCOM_3$) *:*

$LCOM_3(c)$ est similaire à $LCOM_2(c)$ mais utilise la définition suivante des arcs :

$$E_c(c) = \{\ [m_1, m_2] : m_1, m_2 \in V_c(c) \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \Phi \vee m_1 \in SIM(m_2) \vee m_2 \in SIM(m_1)\ \}$$

Lack of Cohesion in Methods ($LCOM_4$) :

$LCOM_4(c)$ est définie par Chidamber et Kemerer comme étant le nombre de paires de méthodes dans une classe n'ayant aucune référence d'attributs communs, $|P|$, moins le nombre de paires des méthodes similaires $|Q|$:

Soit $Q = E_c de\ LCOM_2(c)$ et $P = \begin{cases} 0, si\ AR(m) = 0\ \forall\ m \in M_I(c) \\ condition de\ LCOM_1(c), \text{ sinon} \end{cases}$

$$\text{Alors: } LCOM_4(c) = \begin{cases} 0, si\ |\mathrm{P}| < |Q| \\ |\mathrm{P}| - |Q|, \text{ sinon} \end{cases}$$

Lack of Cohesion in Methods ($LCOM_5$) :

$LCOM_5(c)$ est définie par Henderson (1996) :

$$LCOM_5(c) = \frac{|M_I(c)| - \frac{1}{A_I(c)} \sum_{a \in A_I(c)} |\{m : m \in M_I(c) \wedge a \in AR(m)\}|}{|M_I(c)| - 1}$$

Coh, Lack Of Cohesion in methods:

Cette métrique est une variation de $LCOM_5$. Une valeur de *Coh* élevée indique une cohésion élevée et une faible valeur montre une cohésion basse, elle est définie par :

$$Coh(c) = \frac{\sum_{a \in A_I(c)} |\{m : m \in M_I(c) \wedge a \in AR(m)\}|}{|M_I(c)| - A_I(c)}$$

Co,Connectivity :

Soient V_c et E_c définies dans $LCOM_3$, cette métrique est utilisée dans le cas ou $LCOM_3 = 1$, alors Co est définie comme suit :

$$Co(c) = 2\frac{|E_c| - (|V_c| - 1)}{(|V_c| - 1)(|V_c| - 2)}$$

TCC, Tight Class Cohesion :

TCC est le pourcentage de paires de méthodes publiques d'une classe qui sont connectées directement ou indirectement en utilisant des attributs communs.

$$TCC(c) = \frac{|\{[m_1, m_2] : m_1, m_2 \in M_I(c) \cap M_{Pub}(c) \wedge m_1 \neq m_2 \wedge cau(m_1, m_2)\}|}{|M_I(c) \cap M_{Pub}(c)|(|M_I(c) \cap M_{Pub}(c)| - 1)}$$

Avec :

$$cau(m_1, m_2) = \left(\bigcup_{m \in \{m_1\} \cup SIM^*(m_1)} AR(m) \right) \cap \left(\bigcup_{m \in \{m_2\} \cup SIM^*(m_2)} AR(m) \right) \cap A_I(c) \neq 0$$

Le prédicat $cau(m_1, m_2)$ est vrai, si $m_1, m_2 \in M_I(c)$ utilisent directement ou indirectement un attribut en commun de la classe c.

LCC, Lose Class Cohesion :

LCC est similaire à *TCC,* sauf que *LCC* considère aussi les paires de méthodes indirectement connectées

$$LCC(c) = \frac{\left|\left\{[m_1, m_2]: m_1, m_2 \in M_I(c) \cap M_{Pub}(c) \wedge m_1 \neq m_2 \wedge cau^*(m_1, m_2)\right\}\right|}{\left|M_I(c) \cap M_{Pub}(c)\right|\left(\left|M_I(c) \cap M_{Pub}(c)\right| - 1\right)}$$

ICH, Information flow based Cohesion :

La cohésion d'un ensemble de classes est la somme de la cohésion des classes dans l'ensemble. Elle représente le nombre d'invocations des autres méthodes de même classe.

$$ICH^c(m) = \sum_{m' \in M_{NEW}(c) \cup M_{OVR}(c)} \left(1 + \left|Par(m')\right|\right) NPI(m, m')$$

$$ICH = \sum_{m \in M_I(c)} ICH^c(m)$$

1.4.3.3 Mesures de couplage

Le couplage concerne les relations qu'a un module avec les autres modules du système. Il dépend de la complexité des interfaces entre modules. Intuitivement, un faible degré de couplage entre les modules est souhaitable, car en effet, un module relativement peu relié aux autres est facile à comprendre, à modifier, à tester et enfin à réutiliser. De plus, les chances de propagation des erreurs d'un module à un autre se trouvent d'autant plus diminuées que le nombre de liens entre ces deux modules est petit.

CBO, Coupling Between Object classes:

La définition de cette mesure a été proposée par Chidamber et Kemerer (1994). *CBO* est le nombre de classes avec lesquelles une classe est couplée. Une classe *A* est couplée avec une classe *B* si les méthodes de la classe *A* utilisent les méthodes ou les attributs de la classe *B* ou vice versa. Cela inclut le couplage basé sur l'héritage (couplage entre deux classes reliées par héritage).

$$CBO(c) = \left|\left\{d \in C \setminus \{c\} : uses(c, d) \vee uses(d, c)\right\}\right|$$

CBO', Coupling Between Object classes :

Une petite différente à *CBO, CBO'* ne prend pas en considération le couplage entre les classes reliées par l'héritage.

$$CBO'(c) = \left|\{d \in C \setminus \{\{c\} \cup Ancestors(c)\} : uses(c,d) \vee uses(d,c)\}\right|$$

***RFC, Response For a Class* :**

C'est le nombre de méthodes d'une classe plus le nombre de méthodes invoquées directement ou indirectement dans cette classe. Cette mesure sert à évaluer la testabilité, la complexité. Plus une classe a un *RFC* élevé, plus elle est compliquée à comprendre et plus l'effort de test sera important.

RFC', Response For a Class :

C'est le nombre de méthodes qui peuvent être possiblement invoquées en réponse à un message envoyé à une classe *c*, incluant le nombre de ses méthodes, les méthodes invoquées par les méthodes de celle-ci et ainsi de suite.

$$RFC'(c) = RFC_\alpha(c)$$

$RFC_\alpha(c)$, Response For a Class:

Pour un niveau α spécifié :

$$RFC_\alpha(c) = \left|\bigcup_{i=0}^{\alpha} R_i(c)\right|, \quad R_0(c) = M(c) \quad \text{et} \quad R_{i+1}(c) = \bigcup_{m \in R_i(c)} PIM(m)$$

MPC, Message Passing Coupling :

Cette métrique a été définie par Li et Henry (1993). C'est le nombre d'invocations de méthodes dans une classe.

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

ICP, Information Based Coupling :

Cette métrique a été introduite par Lee et al. (1995), c'est le nombre d'invocations de méthodes dans une classe pondéré par le nombre de paramètres des méthodes invoquées.

$$ICP(c) = \sum_{m \in M_I(c)} ICP^c(m)$$

$$ICP^c(c) = \sum_{m' \in PIM(m) | (M_{New}(c) \cup M_{OVR}(c))} (1 + |Par(m')|) NPI(m, m')$$

IH-ICP, Information Flow Based Inheritance Coupling :

Même chose que *ICP* mais compte les invocations de méthodes des classes ancêtres (couplage par héritage).

$$IH - ICP(m) = \sum_{m \in M_I(c)} IH - ICP^c(m)$$

$$IH - ICP^c(m) = \sum_{m' \in PIM(m) \cap (\bigcup_{c' \in C \setminus \{\{c\} \cup Ancestors(c)\}} M(c'))} (1 + |Par(m')|) NPI(m, m')$$

NIH-ICP, Information Flow Based Non-Inheritance Coupling :

Même chose que *ICP*, mais mesure le nombre d'invocations de classes sans compter le couplage par héritage.

$$NIH - ICP(m) = \sum_{m \in M_I(c)} NIH - ICP^c(m)$$

$$NIH-ICP^{c}(m)=\sum_{m'\in PIM(m)\cap(\bigcup_{c'\in Ancestors(c)}M'(c))}(1+|Par(m')|)NPI(m,m')$$

<u>DAC, Data Abstraction Coupling :</u>

Cette métrique est définie par Li et Henry. C'est le nombre d'attributs dans une classe qui ont une classe comme type (instance d'une classe).

$$DAC(c)=\sum_{d\in C,d\neq c}ACA(c,d)$$

<u>DAC', Data Abstraction Coupling :</u>

Définie par Li et Henry(1993). C'est le nombre des différentes classes qui sont utilisées comme type pour des attributs d'une classe.

$$DAC'(c)=\sum_{d\in C,d\neq c}(ACA(c,d)>0)$$

Les métriques suivantes comptent le nombre d'interaction entre classes. Ces mesures distinguent la relation entre les classes, le locus, et le type (*Briand et al.*, 1997).

- La première ou les deux premières lettres décrivent la relation (A : coupling to ancestor classes, D : descendent, F : friend classes, IF : inverse friends, O : others).
- Les deux lettres suivantes décrivent le type d'interaction entre classes : classe-attribut (CA), classe-méthode (CM) et méthode-méthode (MM).
- Les deux dernières lettres définissent le locus qui peut être de valeur 'import' ou 'export' (IC : import coupling, EC : export coupling).

Voilà les formules de chacune, comme décrites dans les travaux de Mao Y. en 1998.

IFCAIC, Inverse Friend Class-Attribute Import Coupling :

$$IFCAIC(c) = \sum_{d \in Friends^{-1}(c)} ACA(c,d)$$

ACAIC, Ancestors Class-Attribute Import Coupling :

$$ACAIC(c) = \sum_{d \in Ancestors(c)} ACA(c,d)$$

OCAIC, Others Class-Attribute Import Coupling :

$$OCAIC(c) = \sum_{d \in Others(c)} ACA(c,d)$$

FCAEC, Friends Class Export Coupling :

$$FCAEC(c) = \sum_{d \in Friends(c)} ACA(d,c)$$

DCAEC, Descendants Class-Attribute Export coupling :

$$DCAEC(c) = \sum_{d \in Descendants(c)} ACA(d,c)$$

OCAEC, Others Class-Attribute Export Coupling :

$$OCAEC(c) = \sum_{d \in Others(c)} ACA(d,c)$$

IFCMIC, Inverse Friend Class-Method Import Coupling :

$$IFCMIC(c) = \sum_{d \in Friends^{-1}(c)} ACM(c,d)$$

ACMIC, Ancestors Class-Method import coupling :

$$ACMIC(c) = \sum_{d \in Ancestors(c)} ACM(c,d)$$

OCIC, Others Class-Method Import Coupling :

$$OCMIC(c) = \sum_{d \in Others(c)} ACM(c,d)$$

FCMEC, Friends Class-Method Export Coupling :

$$FCMEC(c) = \sum_{d \in Friends(c)} ACM(c,d)$$

DCMEC, Descendants Class-Method export coupling :

$$DCMEC(c) = \sum_{d \in Descendants(c)} ACM(c,d)$$

OCMEC, Others Class-Method Export Coupling :

$$OCMEC(c) = \sum_{d \in Others(c)} ACM(d,c)$$

IFMMIC, Inverse Friend Method-Method Import Coupling :

$$IFMMIC(c) = \sum_{d \in Friends^{-1}(c)} AMM(c,d)$$

AMMIC, Ancestors Method-Method Import Coupling :

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

OMMIC, Others Method-Method Import Coupling :

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

FMMEC, Friends Method-Method Export Coupling :

$$FMMEC(c) = \sum_{d \in Friends(c)} AMM(d,c)$$

DMMEC, Others Method-Method Export Coupling :

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

OMMEC, Others Method-Method Export Coupling :

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

1.4.3.4 Mesures de taille et de complexité

La complexité du logiciel influence énormément sur l'utilisation et la modification d'un composant logiciel. Elle est définie comme étant le degré de difficulté à comprendre ou à vérifier la conception de ce dernier. Elle est considérée comme un indicateur de fiabilité, de réutilisabilité, de maintenance et de compréhension. On trouve plusieurs métriques qui la décrivent :

NOM (Number Of Methods):

Le nombre de méthodes locales dans une classe. Plus NOM est élevée, plus une classe est complexe.

CIS (Class Interface size):

Le nombre de méthodes publiques dans une classe.

NOT (Number of Trivial Methods):

Le nombre de méthodes triviales, qui sont marquées comme incorporées en C++ dans une classe.

NOP (Number of Polymorphic Methods):

Le nombre de méthodes qui peuvent montrer un comportement polymorphique (y compris surchargées),

NOD (Number of Attributes):

Le nombre d'attributs dans une classe.

NAD (Number of Abstract Data Types):

Le nombre d'objet définis par l'utilisateur.

NRA (Number of Reference Attributes):

Le nombre de pointeurs et de références utilisés comme attributs dans une classe.

NPA (Number of Public Attributes):

Le nombre d'attributs déclarés comme publiques dans une classe.

NOO (Number of Overloaded Operators):

Le nombre de méthodes opérateurs surchargées définies dans une classe.

NPM (Number of Parameters Per Method):

Le nombre moyen de paramètres par méthode dans une classe.

CSB (Class Size in Bytes):

La taille en octet des objets qui seront créés par la partie déclaration d'une classe. Elle est égale à la somme des tailles de tous les attributs déclarés.

1.5 Prédiction de la qualité de logiciel

La fiabilité d'un produit logiciel est définie à partir de sa performance opérationnelle qui ne peut être mesurée qu'après une période de son fonctionnement. Ainsi, un recours à la prédiction s'impose pour évaluer la fiabilité du logiciel en cours de développement. Dans ce cas, la fiabilité est prédite à partir des indicateurs qui peuvent être disponibles dès les premières étapes du développement. À l'aide de certaines mesures, les développeurs, peuvent alors entreprendre les actions correctives convenables

Le besoin de prédiction porte aussi sur la qualité, le coût et de la durée de développement. L'importance de la prédiction a attiré l'attention de plusieurs chercheurs qui ont proposé différent méthodes afin de résoudre ce problème par la construction de systèmes de prédiction.

1.6 Conclusion

Nous avons essayé dans ce chapitre de donner une vue globale sur les différents aspects du logiciel. La partie la plus importante de ce chapitre concerne la mesure logicielle; elle joue un rôle important dans le développement du logiciel, afin de gérer le processus de construction tout en contrôlant les coûts de production et en assurant la qualité. Nous avons donc classifié et présenté les mesures impliquées ainsi que leur utilité durant et après le développement logiciel. Les métriques présentées dans ce chapitre seront utilisées pour établir l'existence de liens entre des attributs internes et des facteurs de qualité.

CHAPITRE II

GESTION DES CONNAISSANCES EN GÉNIE LOGICIEL

2.1 Introduction

Depuis des nombreux siècles, et même depuis l'aube de l'humanité, les hommes ont préservé et transmis leurs savoirs à travers les générations. Certains chercheurs de divers domaines ont constaté l'arrivée d'une nouvelle économie dominée par la production, l'échange et le partage de la connaissance (*Zacklad et al.* 2001). Dans cette économie, la gestion des connaissances constitue une clé importante qui permet aux organisations d'améliorer leur performance et d'en obtenir un avantage concurrentiel (Ermine, J.L, 2003).

Dans ce chapitre, nous commençons par définir la connaissance en général. Par la suite, nous introduisons le rôle de la gestion de connaissances en génie logiciel et plus précisément, par rapport au développement logiciel.

2.2 La connaissance

2.2.1 Définition

La connaissance est de plus en plus considérée comme un capital stratégique dans les entreprises et les organisations. La connaissance détenue par une entreprise est la somme totale des expériences et le savoir-faire de ses employés. C'est l'ensemble des notions et des principes qu'une personne acquiert par l'étude, l'observation ou l'expérience et qu'elle peut intégrer à des habilités (Nonaka, 1994).

La connaissance est devenue une ressource importante dans les entreprises et les organisations spécialisées en génie logiciel. Le succès de ces dernières dépend de leurs habilités, d'une part de transformer les connaissances tacites de leurs employés en connaissances bien sauvegardées dans des supports électroniques, et d'autre part, de rendre ces connaissances disponibles pour les autres membres pour accomplir une tâche quelconque ou prendre une décision pertinente.

2.2.2 Classification des connaissances

Nonaka en 1994 distingue deux grands types de connaissances :

- explicites (tangibles) : elles peuvent clairement être articulées, exprimées verbalement ou par écrit, documentées, archivées et codifiées, habituellement à l'aide d'outils informatiques. Elles se présentent sous forme de livres ou documents, de rapports, de manuel, de diagrammes, etc.

- tacites (intangibles) : elles sont plus difficile à saisir et à appréhender. C'est le savoir-faire intangible, non codifié, qui est profondément ancré dans les personnes et qui se manifeste en réponse à une situation ou à une action et lors d'une interaction des collègues, des clients et des personnes. La connaissance tacite réside dans l'esprit même des personnes et émerge à travers la communauté. L'acquisition de ce type de connaissances n'est pas une tâche facile. Il peut être très difficile pour un expert de bien exprimer ses connaissances sur papier, spécialement lorsque cette expérience provient de sensations et de la pensées.

2.2.3 La création de la connaissance

La théorie de la création de la connaissance établie par Nonaka (1991, 1994) et popularisée par l'ouvrage de Nonaka et Takeuchi (1995) est sans doute l'un des modèles qui a le plus influencé le management des connaissances.

Le modèle de création des connaissances repose sur la distinction entre savoir tacite et savoir explicite. La connaissance est convertie selon quatre modes :

- ✓ la socialisation : elle représente les différents échanges d'informations, de données ou de connaissances entre les individus partageant les mêmes objectifs professionnels dans l'accomplissement de leurs tâches. Elle repose sur une transmission de connaissances tacites d'un individu à l'autre sans utiliser un langage mais via l'observation, l'imitation et la pratique. Il s'agit du passage des connaissances tacites à tacites.

- ✓ l'externalisation : c'est le transfert des connaissances tacites d'un individu, souvent expérimenté, vers un groupe d'individus. Il s'agit du passage tacite explicite.

- ✓ la combinaison : après la transformation des connaissances tacites en connaissances explicites, ces dernières doivent être diffusées au sein de l'organisation et combinées à d'autres connaissances préexistantes ou nouvellement externalisées. La combinaison est un processus qui se charge de sauvegarder les connaissances des différents groupes (équipes, sous-équipes). Il s'agit du passage explicite explicite.

- ✓ l'internalisation : c'est l'appropriation des connaissances organisationnelles par l'individu par le biais du processus d'apprentissage. Chaque acteur passe par le même cycle d'acquisition et de partage de connaissances, ce qui résout le problème de perte des informations précieuses produites au cours du projet.

Transformation	**Tacite**	**Explicite**
Tacite	Socialisation	Externalisation
Explicite	Internalisation	Combinaison

Tableau 2.1 Modèle de partage de connaissances (*Nonaka*, 1994)

Ainsi, après avoir exploré la littérature sur ce qu'est la connaissance, ses différentes formes et ses principaux modes de transformation, il convient de s'intéresser au concept de gestion des connaissances en génie logiciel afin de prédire ou d'estimer la qualité du produit logiciel.

2.3 La gestion des connaissances

La gestion des connaissances peut être définie comme étant l'exploitation et l'utilisation organisée des savoirs tangibles et intangibles contenus dans une entreprise dans le but d'améliorer sa performance et d'atteindre ses objectifs.

La gestion des connaissances, en termes simples, se base sur quatre composants principaux :

- le contenu ou la recherche de l'information : c'est le fait d'accéder à l'information ou à la connaissance immédiatement et facilement.

- la collaboration ou le partage de l'information ou de la connaissance : c'est le fait de travailler en équipe et d'échanger l'information instantanément pour produire quelque chose.

- l'expertise : c'est de trouver et de travailler avec des experts dans l'entreprise et tirer parti de leurs connaissances et de leurs expériences.

- l'apprentissage : c'est la formation sur le terrain pour développer les carrières et les compétences.

2.3.1 Le rôle de la gestion des connaissances en génie logiciel

Voici quelques raisons qui poussent les compagnies de génie logiciel à faire de la gestion des connaissances:

- La taille et la complexité des produits logiciels.

- L'évolution de l'environnement contraint celles-ci à être beaucoup plus performantes et pouvoir mutualiser et capitaliser les expériences pour ne pas répéter les mêmes efforts lors de chaque nouveau projet.

- Les connaissances cruciales de l'entreprise ne résident pas essentiellement dans les systèmes d'information automatisés qui exploitent les informations structurées et les règles de gestion explicites associées. Les connaissances cruciales sont de plus en plus tacites.

- La perte des compétences liée à des départs à la retraite.

- La prise de conscience que l'on entre dans une phase de surinformation et de la valeur du capital immatériel que représentent aussi bien la connaissance sur les clients que les savoir-faires et les compétences de leurs employés.

2.3.2 Comment gérer le capital de connaissances

Manuel Zacklad et Mickel Grundstein (2001) ont insisté sur le fait que la capitalisation des connaissances est une problématique permanente. Elle s'articule autour de cinq processus clés inter-reliés : le repérage, la préservation, la valorisation, l'actualisation des connaissances et le dernier processus concerne les interactions entre les quatre premiers processus.

Le premier processus liés au repérage des connaissances dans les organisations consiste à développer des nouvelles connaissances à partir des connaissances explicites et tacites de l'organisation.

Le deuxième processus est celui de la préservation des connaissances qui consiste à rendre exploitable les connaissances acquises et créées par une organisation. Le but principal de la préservation des connaissances est de réutiliser et exploiter ces connaissances dans les futurs projets. La préservation consiste à acquérir, modéliser, formaliser et conserver les connaissances par le fait de rédiger des documents, stocker des informations structurées dans des bases de données et aussi élaborer de règles de décision pour des systèmes experts.

La valorisation des connaissances c'est de rendre accessible la connaissance stockée au sein de l'entreprise. Elle représente un échange de connaissances entre une source et une destination. La source peut être des individus détenant une connaissance ou bien des bases de données où sont stockées les connaissances et la destination peut être d'autres individus qui en ont besoin.

Le quatrième processus concerne le problème lié à l'actualisation des connaissances : il faut les évaluer, les mettre à jour, les standardiser et les enrichir avec création de toute nouvelle connaissance.

Enfin, le dernier processus est celui de la gestion de la connaissance; il représente les interactions entre les différents problèmes cités précédemment. Son but principal est de sensibiliser, former, motiver tous les acteurs dans une organisation, organiser et coordonner les activités, et encourager le partage des connaissances.

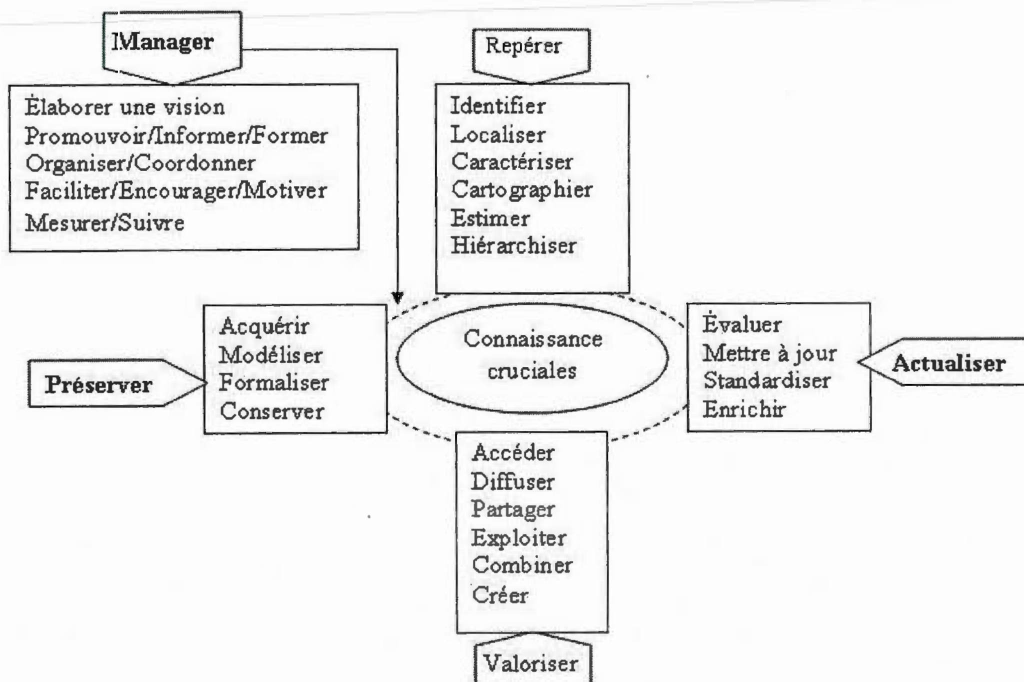

Figure 2.1 Capitalisation des connaissances (Manuel Zacklad et Mickel Grundstein)

2.3.3 Réutilisation des connaissances en génie logiciel

Plusieurs travaux de recherche ont porté sur la réutilisation des connaissances en général et particulièrement la réutilisation de connaissances dans le domaine du développement logiciel.

Plusieurs études réalisées sur la réutilisation logicielle au sein des entreprises démontrent la nécessité de fournir des outils performants pour mieux gérer l'expérience existante (*Nguyen et al.*, 1997).

En plus des avantages de réduction de coûts et d'amélioration de la productivité, la gestion des connaissances et la réutilisation de l'expérience acquise au cours du développement logiciel aide fortement à l'amélioration de la qualité des systèmes développés (Basset, 1987, Basili et Rombach, 1994, Griss, Favaro et Falton, 1994).

2.3.4 Le rôle de la gestion des connaissances dans un système d'aide à la décision

Un système d'aide à la décision (SAD) vise à aider et accompagner l'humain (expert ou autre) dans ses décisions. Ce type de système peut proposer à son utilisateur des bonnes solutions et des bonnes propositions afin de prendre la bonne décision et faire de bons raisonnements mais il lui laisse toujours le libre choix de prendre lui-même la décision finale. Nous trouvons les SAD dans plusieurs domaines et en particulier dans les organisations de génie logiciel. Un SAD aide à concevoir des logiciels de bonne qualité en réduisant la quantité d'effort et de temps.

L'élément essentiel d'un SAD est la base de connaissances, dans laquelle se trouvent toutes les informations et les connaissances spécifiques sur un domaine particulier. Contrairement à un humain, un SAD peut conserver dans sa mémoire une grande quantité de connaissances. Ces connaissances doivent être de bonne qualité et bien organisées. Les connaissances peuvent être sous la forme de règles. Ces dernières peuvent être jumelées à un moteur d'inférence (moteur à base de règles comme JRules que nous verrons plus loin) afin d'appliquer un raisonnement. Le moteur d'inférence parcourt la base de connaissances en décidant quelles règles il doit analyser, quelles règles il doit éliminer et lesquelles

correspondent bien au problème pour donner un résultat pertinent et logique. Ainsi, les bénéfices de l'utilisation des SAD à base de connaissances sont nombreux :

- augmenter la productivité et améliorer les performances des entreprises.
- augmenter la rapidité dans la prise de décision.
- améliorer la consistance et l'efficacité concernant la précision de la qualité des décisions et des prédictions.

2.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la gestion des connaissances comme une solution stratégique dans les organisations de génie logiciel. L'acquisition des connaissances est une activité critique dans l'élaboration d'une bonne base de connaissances, et par conséquent, d'un bon système d'aide à la décision pour la prédiction et l'estimation de la qualité logicielle. Tout comme l'expert humain qui a besoin de bonnes connaissances pour bien raisonner, un SAD en a lui aussi besoin.

CHAPITRE III

UNE APPROCHE POUR L'ACQUISITION DES CONNAISSANCES EN GÉNIE LOGICIEL

3.1 Introduction

Pendant ces dernières années, un grand nombre de modèles de prédiction de la qualité des logiciels a été proposé. Le but principal de ces méthodes est de prédire un facteur de qualité (maintenabilité, réutilisabilité, etc.) à partir d'un ensemble de mesures directes que sont les métriques.

En général, deux types de méthodes sont particulièrement bien documentés dans la littérature, pour gérer les connaissances en génie logiciel et pour construire ces modèles prédictifs de qualité : la première méthode repose sur l'expertise humaine qui se présente sous forme d'heuristique. La seconde est basée sur l'apprentissage automatique des données historiques contenues dans la base de connaissances de l'entreprise. Le but principal de ces deux méthodes est d'améliorer la prédiction de la qualité du produit logiciel et d'augmenter la pertinence et l'utilité de ces modèles prédictifs.

Un problème qui se pose souvent dans le domaine de génie logiciel est le manque de données qui rend les modèles de prédiction difficiles à généraliser, à valider et à utiliser.

Le choix d'un modèle de prédiction de la qualité est une décision difficile et non triviale puisque les modèles universels de prédiction de la qualité n'existent pas encore. La pénurie de données ne fait qu'accentuer ce problème. Plusieurs raisons sont à l'origine de cette pénurie de données. Nous relevons par exemple la rareté, voir l'absence, des collectes

systématiques de données dans les compagnies de logiciels ou encore la confidentialité de l'information.

3.2 L'expertise humaine

Un expert est souvent un ex-développeur qui a acquis au fil des années une certaine expérience lui permettant d'asseoir son expertise. Cette expérience doit couvrir dans l'idéal une variété de plates-formes et de produits, et une connaissance de la vie d'une application, voire d'un projet. Il a un rôle clef au sein d'une équipe de développement. L'expert doit travailler main dans la main avec tous les membres de groupe afin d'obtenir les meilleures résultats.

C'est par le biais d'échanges permanents avec les développeurs et les chefs de projet que l'expert pourra communiquer ses connaissances, mais aussi surtout tirer le meilleur du travail de l'équipe. L'expert doit également être à l'écoute des besoins. Il doit soumettre des propositions pour l'orientation des développements avec une vision sur le long terme de l'architecture du système d'information et des problématiques d'intégration des applications, mais tout en gardant bien en tête les priorités. En effet, il peut être amené à écrire des lignes et valider la qualité de ce qui est produit ou décoincer une situation techniquement difficile. Cela signifie donc, surtout, qu'il doit être capable de lire et analyser du code de manière très rapide.

Un expert peut aussi proposer sa connaissance sous forme de règles. Ces règles sont par la suite transformées en un module informatique. Le module met donc en application les règles de l'expert pour résoudre un problème.

3.3 L'apprentissage automatique

Notre objectif dans cette section est de présenter quelques techniques d'apprentissage automatique utilisées pour l'estimation et la prédiction de la qualité de logiciel.

3.3.1 Définition

Parmi les nombreuses définitions proposées dans la littérature, nous avons retenu une que nous trouvons très claire

C'est une définition fournie par Mitchell en 1986: « A learning process can be view as a transformation of information provided by a source of information (a teacher or environment) into a more efficient and/or more general form in which it is stored for future use. This transformation involves the background knowledge of the learner (that includes goals of learning, concepts learned from the past experience, hypothesis evaluation criteria, etc.) and uses various types of inference. The type of inference employed in any given act of learning depends on the source of information and the goal of learning....For example, when learning a concept from examples, explaining the behavior of a system, or discovering an equation ».

Cette définition décrit l'apprentissage comme un processus de transformation de données impliquant les connaissances et les stratégies d'inférence.

Les algorithmes d'apprentissage sont de plus en plus utilisés pour la modélisation de la qualité du logiciel et pour compléter éventuellement des opinions d'experts. Les modèles produits sont utilisés pour prédire le coût et l'effort de développement et d'autres facteurs représentant la qualité du logiciel.

Plusieurs travaux ont été menés concernant la prédiction des attributs de qualité en utilisant l'apprentissage automatique. Citons à titre d'exemple les travaux de :

- Porter and Selby en 1990, ont utilisé l'algorithme d'apprentissage automatique ID3 pour construire un modèle prédictif qui permet de détecter, au début du cycle de vie du produit, les composants qui ont un risque élevé à contenir des erreurs et qui peuvent engendrer un coût très élevé.
- H.Lounis et al. en 1999, ont aussi étudié les algorithmes d'apprentissage et leurs capacités à estimer la défectuosité de composants logiciels.

- Boukadoum et al. en 2001, ont utilisé l'apprentissage automatique et la logique floue pour la prédiction de l'évolutivité du logiciel.

3.3.2 Apprentissage supervisé et non supervisé

L'apprentissage supervisé utilise des exemples étiquetés par une classe. Ces étiquettes ou ces classes peuvent être vues comme fournies par un superviseur ou un professeur, d'où le nom d'apprentissage supervisé. La tâche principale de ce type de classification est d'établir une fonction de classification, appelée hypothèse ou règle, permettant de trouver la classe de chaque nouvel exemple. Il réalise donc un raisonnement inductif en passant des exemples particuliers à une fonction de classification générale.

Dans l'apprentissage supervisé, nous allons distinguer deux types d'exemples : les exemples d'apprentissage qui servent à apprendre une règle de classification et les nouveaux exemples à classer qui sont les objets sur lesquels les règles seront appliquées afin de déterminer leurs classes.

L'apprentissage supervisé est très utile pour prédire la classe de nouveaux exemples non encore classés. Elle est aussi utile pour expliquer les règles apprises et comprendre ce qui relie les exemples à leurs classes.

Par contre, l'apprentissage non supervisé repose seulement sur ses entrées. Son but est de découvrir une relation pouvant exister entre les attributs et non la prédiction d'une sortie en fonction des entrées. Cet apprentissage s'occupe donc de la formation automatique de classes. Il existe différents types d'apprentissage non supervisé. Nous pouvons citer trois types qui sont les plus populaires :

- Apprentissage de sous-classes par clustring : on arrange les exemples similaires en agrégats qui caractérisent les différentes classes;
- Apprentissage par expérimentation : le système d'apprentissage peut lui-même générer des exemples pour avancer l'apprentissage;

- Algorithmes génétiques qui permettent l'apprentissage et la découverte d'une structure qui satisfait un critère d'évaluation donné.

3.3.3 Les différentes techniques de l'apprentissage supervisé

Les algorithmes de classification supervisés tentent de trouver un modèle qui explique le lien entre des données d'entrée et les classes de sortie. Il existe de nombreuses méthodes de classification supervisée :

- K plus proches voisins ;
- Arbres de décisions ;
- Apprentissage de règles ;

Nous présenterons brièvement les techniques d'apprentissage les plus populaires que sont les arbres de décision et les règles.

3.3.4 Apprentissage d'arbres de décision

Les arbres de décision sont essentiellement des systèmes de classification qui permettent de prédire le résultat d'un nouvel événement en se basant sur des exemples passés. Le modèle est construit à partir d'un ensemble d'exemples d'apprentissage et est représenté sous la forme d'un arbre.

La classification par les arbres de décision se fait de manière hiérarchique. Ces méthodes sont basées sur la partition récursive de l'ensemble d'apprentissage de façon à construire des sous-ensembles homogènes et qui appartiennent à la même famille (Paul E, Utgoff).

Dans les arbres de décision, nous trouvons un point d'entrée qui est constitué par le premier test, formant ainsi la racine de l'arbre. Après, on fait d'autres tests qui font diriger l'ensemble des individus vers plusieurs branches qui se subdivisent à leurs tours grâce à d'autres tests. Chaque point de connexion entre deux branches s'appelle un nœud.

Les nœuds terminaux sont dénommés des feuilles. Elles portent la prédiction finale et la classe à laquelle appartient chaque individu. Cette classification est facilement compréhensible par un utilisateur car les arbres de décision représentent graphiquement un ensemble de règles aisément interprétables. Nous pouvons interpréter un arbre de décision comme étant une forme normale conjonctive : chaque chemin commençant par la racine menant à une feuille peut s'interpréter comme une conjonction de valeurs d'attributs. Toutes les branches de l'arbre qui mènent vers des feuilles dont les classes sont identiques peuvent s'interpréter comme une disjonction.

On peut donc facilement extraire des règles sous forme normale disjonctive, qui sont non ambiguës et faciles à lire mais peuvent être un peu complexes selon la profondeur de l'arbre. Ce problème peut être résolu par des méthodes de simplification afin d'éliminer les risques de sur-apprentissage et de simplifier ces règles.

Les algorithmes d'induction d'arbres de décision les plus connus sont :

ID3: l'algorithme ID3 fut proposé par Quinlan en 1979 afin de générer des arbres de décision à partir de données. C'est un algorithme basique facile à implémenter dont la première fonction est de remplacer les experts dans la construction d'un arbre de décision. Les principales idées sur lesquelles repose ID3 sont les suivantes :

- Dans l'arbre de décision, chaque nœud correspond à un attribut non cible et chaque arc à une valeur possible de cet attribut. Une feuille de l'arbre donne la valeur escomptée de l'attribut cible pour l'enregistrement testé décrit par le chemin de la racine de l'arbre de décision jusqu'à la feuille.

- Dans l'arbre de décision, à chaque nœud doit être associé l'attribut non cible qui apporte le plus d'information par rapport aux autres attributs non encore utilisés dans le chemin depuis la racine.

- La fonction d'entropie, qui était introduite par Claude Shannon lors de ses recherches concernant la théorie de l'information et qui sert dans plusieurs méthodes du data

mining, est utilisée pour mesurer la quantité d'information apportée par un nœud. L'algorithme ID3 ne peut pas traiter les enregistrements incomplets. De plus, les attributs sont discrétisés, ce qui n'est pas toujours une solution acceptable. Enfin, l'arbre produit peut comporter des sous-arbres dans lesquels on ne va presque jamais. Les arbres de décision ne sont ni robustes, ni compacts ce qui les rendent inadaptés aux grosses bases de données.

C4.5 : Cet algorithme a été proposé en 1993 par Ross et Quinlan, pour pallier les limites de l'algorithme ID3. L'algorithme C4.5 repose sur les mêmes principes que l'algorithme ID3 mais il présente plus d'avantages que ce dernier.

C4.5 est une amélioration d'ID3 qui permet de travailler à la fois avec des données discrètes et des données continues. Il permet également de travailler avec des valeurs d'attribut absentes. Enfin, C4.5 élague l'arbre construit afin de supprimer les règles inutiles et de rendre l'arbre plus compact.

CART (Classification and Regression Tree) : décrit par Breiman et al. à 1984, est basé sur la recherche de partitions dichotomiques. Chaque nœud comporte donc un test logique simple, basé sur un attribut unique, qui conduit à deux branches correspondant aux exemples positifs ou négatifs en fonction des résultats de ce test. Il peut être utilisé aussi bien pour des variables cibles numériques ou qualitatives. L'algorithme arrête la construction de l'arbre lorsqu'il obtient des groupes purs (tous les éléments appartiennent à la même classe).

3.3.5 Apprentissage automatique de règles

Le processus de prédiction de la valeur de l'attribut de sortie d'un cas se réalise en utilisant une méthode (ou un modèle) de prédiction. Plusieurs méthodes de prédiction sont utilisées, comme les systèmes de règles, les systèmes de régression, les réseaux de neurones, etc. La construction d'un modèle nécessite une collection d'observations pour lesquelles les valeurs de sortie sont déjà connues, on les appelle, données d'entraînement (training set), données d'apprentissage ou encore données de construction.

Dans le cadre de notre travail, Nous nous intéressons aux méthodes de classification par apprentissage de règles (Mitchell Tom, 1997) qui entre dans le cadre de l'apprentissage automatique supervisé. Les règles que nous cherchons à générer sont de la forme :

Si < *condition* > alors < *classe* >.

La classe prédite par la règle correspond bien à la classe majoritaire de l'ensemble des exemples couvert par la condition de la règle. Ensuite, la disjonction de ces règles permet de couvrir tout l'ensemble d'apprentissage. La prédiction de la classe d'un nouvel exemple se fait par la recherche d'une règle dont la condition satisfait la description de cet exemple. Les structures de ces règles ont l'avantage d'être considérées comme des présentations très expressives et plus faciles à comprendre par l'humain.

3.3.5.1 Règles propositionnelles et règles en logique du premier ordre

Dans le domaine de l'apprentissage automatique supervisé de règles, nous pouvons distinguer deux types de règles (Mitchell, 1997). Certaines techniques produisent des règles propositionnelles d'ordre zéro et d'autres techniques, des règles en logique du premier ordre. Les règles propositionnelles sont les plus simples.

Règles propositionnelles :

Étant donné un ensemble d'exemples E, décrit sur un ensemble d'attributs $A = \{a_1, a_2, ..., a_n\}$, numériques ou symboliques et étiquetés par une classe C(e), nous souhaitons apprendre à prédire la classe d'un nouvel exemple **e'**. Une règle en logique propositionnelle d'ordre zéro est de la forme : If *<cover>* then predict *<class >*, où *cover* est une disjonction de complexes. Un complexe est une conjonction de sélecteurs c'est à dire un test sur un attribut de la forme a_i # v avec a_i un attribut de A, v appartient au domaine de valeurs de a_i, et # est un opérateur de l'ensemble $\{=, \neq, \leq, \geq, <, >\}$ (*Clark et al.*, 1991)

Nous disons qu'un *complexe* couvre un exemple, si le complexe (resp. un *sélecteur*) est vrai pour cet exemple. Ainsi, un complexe est vide (conjonction de zéro sélecteur) s'il couvre tous les exemples de l'ensemble E.

L'inconvénient principal de ces règles propositionnelles est qu'elles ne font pas référence à des variables génériques, Elles sont très spécifiques aux exemples de l'ensemble d'apprentissage.

Règles en logique du premier ordre :

Les règles en logique de premier ordre ont été proposées dans l'apprentissage de règles pour pallier les limites des règles propositionnelles en permettant la représentation de variables et des relations d'une façon plus générale et non spécifique (*Clark et al.*, 1991). L'apprentissage de règles du premier ordre est aussi connu sous l'appellation d'ILP 'inductive logic programming'.

La programmation logique inductive permet la construction, à partir d'un ensemble d'exemples, d'une hypothèse avec des variables. Une hypothèse est un ensemble de clauses de Horn, c'est-à-dire un programme défini, qui satisfait tous les exemples positifs et ne satisfait pas les exemples négatifs. Nous disons dans ce cas que cette hypothèse couvre les exemples. Une clause est définie comme étant une disjonction de littéraux positifs ou négatifs. Une clause de Horn du premier ordre est une clause dont au plus un de ses littéraux est positif, les autres sont négatifs. Un littéral est un atome (ou la négation d'un atome), de la forme $p(t_1,\ldots, t_n)$ ou p est un prédicat et où les t_i représentent des variables ou des constantes. La partie gauche dans la règle désigne la tête (ou le conséquent) tandis que la partie droite désigne le corps de la clause (ou les antécédents). Le corps d'une clause est constitué par la conjonction de littéraux (positifs) et la tête d'un littéral unique.

Les techniques d'apprentissage de règles du premier ordre sont plus expressives et permettent de traiter des problèmes difficiles voir impossible à représenter dans le formalisme propositionnel. Les clauses logiques permettent également une définition récursive d'un concept, ce qui est impossible dans le formalisme propositionnel. C'est ainsi le cas de l'algorithme FOIL (*Quinlan et al.*, 1993)

Comme nous l'avons vu plus haut, ces deux formalismes constituent les deux principales catégories de méthodes d'apprentissage de règles. Dans ce qui suit, nous allons présenter quelques algorithmes pour chacune de ces catégories.

3.3.5.2 Algorithmes de couverture

Les techniques d'apprentissage de règles sont basées généralement sur des algorithmes de couverture. Il existe deux types d'algorithmes de couverture qui sont :

- Séquentiel : apprend une hypothèse à la fois.
- Simultané : apprend plusieurs hypothèses en parallèle comme dans les arbres de décision.

L'origine de la stratégie de couverture est due à Michalski (1969) (Algorithme AQ). Tous les algorithmes appelés « *separate-and-conquer* » ou « *covering* » (correspondant à la procédure Séquentiel-Covering présenté par Mitchell en 1997) partagent le même principe :

- Chercher une règle qui couvre une partie des exemples positifs.
- Enlever les exemples positifs couverts de la base d'apprentissage initiale.
- Recommencer récursivement le processus jusqu'à ce qu'il n'y ait plus d'exemples positifs à couvrir.

À la fin de ce processus, chaque exemple de la base d'apprentissage est couvert par au moins une règle. La construction d'une règle dépend du choix des littéraux (attribut, relation, valeur) et du test d'arrêt. Pour l'ajout d'un littéral à une règle, nous pouvons prendre le meilleur littéral au sens d'un critère (ex. l'entropie). L'arrêt peut se faire en fonction d'un critère numérique lié au taux d'erreur. Ces algorithmes fournissent des règles similaires à celles d'ID3.

Une règle de bonne précision est une règle qui ne commet pas d'erreurs, c'est à dire qui ne couvre pas d'exemples négatifs; nous disons qu'une règle est de couverture quelconque lorsqu'elle couvre un certain nombre d'exemples positifs mais pas forcément un grand nombre.

- Procedure Separate-and-Conquer : voici l'algorithme de la procédure tel que décrit par Fürnkranz en 1999.

```
Procedure SeparateAndConquer (EXamples)
Theory = {}
While Positive (Examples) ≠ {}
        Rule = FindBestRule (Examples)
        Covered = Cover (Rule, Examples)
        If RuleStoppingCriterion (theory, Rule, Examples)
                Exit while
        Examples = Examples \ Covered
        Theory = theory ∪ Rule
Theory = PostProcess (theory)
Return (theory)
```

Cette procédure commence par initialiser à vide la variable *theory* qui représente l'ensemble de toutes les règles construites. Ensuite, tant que l'ensemble des exemples positifs n'est pas vide, l'algorithme fait appel à la sous-procédure *FindBestRule* pour chercher la meilleure règle qui couvre le plus de ces exemples positifs. Il identifie l'ensemble des exemples couverts par cette règle par la sous-procédure *Cover*, il vérifie ensuite un critère d'arrêt avec la sous-procédure *RuleStoppingCriterion*, il enlève les exemples couverts de l'ensemble global des exemples et il ajoute la règle à l'ensemble theory. Enfin, il applique un traitement sur l'ensemble theory afin d'affiner les règles trouvées.

- Sous Procédure pour la recherche de la meilleure règle :

```
Procedure FindBestRule (Examples)
InitRule = InitilizeRule (Examples)
InitVal = EvaluateRule (InitRule)
BestRule = < InitVal, InitRule >
Rule = {BestRule}
While Rule ≠ {}
      Candidates = SelectCandidates (Rules, examples)
       Rules = Rules \ Candidates
       For Candidates ∈ Candidates
              Refinement = RefineRule (Candidate, Examples)
              For Refinement ∈ Refinements
                          Evaluation = EvaluateRule (Refinement, Examples)
                          Unless StoppingCriterion (Refinement, Evaluation, examples)
```

```
                                   NewRule = < Evaluation, Refinement>
                                   Rules = Insertsort (NewRule, Rules)
                                   If newRule > BestRule
                                           NewRule =BestRule
        Rules = FiltrerRules (Rules, Examples)
Return (BestRule)
```

Le but principal de la procédure *FindBestRule* est de chercher la meilleure règle à partir d'un critère d'évaluation de ses hypothèses. La valeur de cette heuristique est élevée si la règle candidate couvre beaucoup d'exemples positifs et peu d'exemples négatifs.

La procédure *FindBestRule* se fait selon ces étapes :

- ✓ instancier une règle initiale par la sous-procédure InitilizeRule;
- ✓ évaluer cette règle selon un critère de recherche (ex : gain d'information ou entropie);
- ✓ sauvegarder la valeur de cette évaluation et la règle associée dans une variable BestRule.
- ✓ effectuer une boucle sur la liste des règles :
    1. sélectionner un sous-ensemble de règles candidates.
    2. éliminer ces règles de la liste globale.
    3. effectuer une suite de raffinements sur chaque règle de ce sous-ensemble de règles et évaluer chaque raffinement avec la sous-procédure *EvaluateRule*.
    4. mettre la règle dans son bon emplacement dans la liste des règles selon la valeur de son évaluation.
- ✓ retourner la meilleure règle.

3.3.5.1 Biais utilisés pour l'apprentissage de règles

L'apprentissage de règle est un mécanisme d'induction, c'est à dire le passage du particulier au général. Afin de résoudre ce problème, les algorithmes inductifs utilisés lors de l'apprentissage font des suppositions raisonnables sur la définition des hypothèses reliant les exemples et leurs classes. Cette vision de l'apprentissage est liée à la notion des biais d'apprentissage.

La définition d'un tel biais est une condition nécessaire au bon fonctionnement de la méthode (Mitchell, 1980). Ce biais formule une restriction ou une préférence sur l'ensemble des apprentissages possibles, guidant ainsi la recherche.

Il existe trois principaux types de biais selon Johannes Fürnkranz : l***es biais de langage*** qui définissent le langage des hypothèses recherchées, ***les biais de recherche*** qui orientent le parcours de l'espace de recherche en terme d'élagage et de création de nouvelles hypothèses et les ***biais de validation*** qui englobent les critères d'arrêt de la recherche.

- Les biais de langage

Ils définissent le langage dans lequel sont exprimés les concepts qu'on veut chercher à apprendre. Ainsi, en choisissant un langage adapté aux concepts recherchés, nous restreignons l'espace des concepts possibles. Par exemple, le langage des exemples peut être un langage lié aux règles propositionnelles attributs/valeurs ou un langage plus expressif comme les langages de la logique du premier ordre.

- Les biais de recherche

Les biais de recherche déterminent dans quelle partie de l'espace des hypothèses on doit mener la recherche et de quelle façon on doit rechercher dans cet espace d'hypothèses.

Il y a beaucoup d'option de recherche afin de fouiller l'espace d'apprentissage. Il y a différents algorithmes de recherche : hill-climbing (escalade de colline), beam search (recherche en faisceau), best first search (recherche du meilleur en premier) et la recherche stochastique. Il y a aussi différentes stratégies de recherche telle que la recherche ascendante et la recherche descendante. Enfin, il y a différentes heuristiques de recherche qui servent en général à chercher les meilleures règles avec meilleures qualités.

La recherche basée sur l'algorithme de type 'Hill-Climbing' est la plus utilisée. Plusieurs systèmes l'utilisent tel que AQ, CN2 et FOIL.

Cette méthode de recherche en profondeur utilise une fonction heuristique pour choisir à chaque étape la meilleure règle, mais elle comporte quelques inconvénients :

1. Les fonctions d'évaluation peuvent être trompeuses.
2. Si on arrive à un maximum local de la fonction d'évaluation (une hypothèse dont l'évaluation est supérieure à tous ses successeurs), il n'y a plus de choix possibles et l'algorithme s'arrête, sans avoir trouvé la solution.

La recherche 'beam search' explore en parallèle une liste d'hypothèses candidates. L'avantage principal de ce type de recherche est de réduire la myopie relative à la recherche 'hill-climbing'.

La stratégie du meilleur d'abord 'Best first' consiste à sélectionner l'hypothèse la plus prometteuse par rapport à l'heuristique, parmi toutes les hypothèses rencontrées depuis le début.

Il existe principalement trois types de stratégies de recherche : la recherche *top down,* la recherche *bottom up* et la recherche mixte qui combine les deux approches précédentes. Les algorithmes *top down* partent de la clause la plus générale et la spécialise pas à pas alors que les procédures *bottom up* partent quant à elles d'un fait et elles le généralisent.

Les biais de recherche reliés aux heuristiques d'évaluation permettent d'estimer la qualité des règles trouvées et de guider les algorithmes vers les bonnes régions dans l'espace de recherche. La plupart de ces heuristiques peuvent être implémentées dans la sous procédure EvaluateRule.

En général, les heuristiques de recherche utilisées dans la procédure 'Separate-and-conquer' sont similaires aux heuristiques utilisées dans d'autres algorithmes d'apprentissage inductifs tels que les arbres de décision.

Toutes les heuristiques tentent à déterminer les propriétés de base de chaque règle candidate comme par exemple le nombre d'exemples positifs et le nombre d'exemples

négatifs qu'elle couvre. Il existe différents types d'heuristique d'évaluation de règle. Le premier type est celui des heuristiques de base qui permet de mesurer la couverture relative de la règle telles que :

- ✓ ***Accuracy*** qui permet d'évaluer l'exactitude d'une règle r et elle calcule le pourcentage des exemples correctement classifiés.
- ✓ ***Purity*** qui permet d'évaluer les règles avec leurs puretés. Cette mesure atteint sa valeur optimale lorsqu'aucun exemple négatif n'est couvert par la règle.
- ✓ ***Entropie*** mesure l'homogénéité d'un ensemble d'exemples vis à vis d'un ensemble de classes que l'on cherche à caractériser. Si tous les exemples appartiennent à une seule classe, l'entropie est égale à 0. L'entropie est utilisée dans les premières versions du système d'induction de règles CN2 (*Clark et al.*, 1997).

Un deuxième type d'heuristiques pour évaluer les règles permet d'estimer la complexité, comme par exemple :

- ✓ ***Longueur de la règle*** : qui permet de compter le nombre de ses conditions.
- ✓ ***Couverture positive*** : consiste à déterminer la règle la plus courte, la plus générale et qui couvre plusieurs exemples positifs.
- ✓ ***Gain d'information pondéré*** : cette heuristique est utilisée par le système FOIL. Elle est basée sur la différence des contenus d'information des règles et sur une pondération de cette différence avec le nombre d'exemples positifs.

- Les biais de validation

On dit qu'une hypothèse couvre trop (overfit) les exemples, quand elle est trop spécifique aux exemples de l'ensemble d'apprentissage. Dans le cas de données bruitées, l'overfiting se traduit par la création d'hypothèses trop fines, adaptées que pour classer les exemples bruités, alors que ceux-ci devraient être négligés. Le but d'utiliser ce type de biais est d'éviter la présence de ce genre d'hypothèses qui ne peuvent classer que les exemples de l'ensemble d'apprentissage. Ainsi, les biais de validation définissent un critère d'acceptation pour le

système d'apprentissage, indiquant au système quand la recherche pour la règle considérée doit s'arrêter.

3.3.6 Quelques systèmes d'apprentissage de règles

3.3.6.1 Le système AQ

Nous présentons dans cette section, le principe général du système AQ (*Clark et al.*, 1991). Ce système fonctionne par apprentissage de règles pour chacune des classes. Il utilise lors de son apprentissage l'algorithme de couverture séquentiel et il construit chaque règle par l'approche du plus général au plus spécifique.

Lors de la classification, à chaque étape, il ne considère qu'une seule classe. Pour chaque classe, il divise l'ensemble d'exemples d'apprentissage en deux sous-ensembles. Les exemples positifs pour ceux qui appartiennent à cette classe et les exemples négatifs pour le reste. Puis, il exécute l'algorithme de couverture sur le sous-ensemble d'exemples positifs.

A chaque fois, il génère une nouvelle règle et les exemples positifs couverts par cette règle seront supprimés de l'ensemble. Il répète ce processus jusqu'à ce que chaque exemple soit couvert par au moins une règle

3.3.6.2 Le système CN2

L'algorithme de couverture CN2 est un algorithme de couverture séquentielle qui est assez proche du système AQ (*Clark et al.*, 1991) : les règles sont générées séquentiellement par une méthode du plus général au plus spécifique et cet apprentissage séquentiel des règles pour chaque classe se fait par un parcours de l'espace de recherche en faisceau 'beam search'.

Initialement, l'ensemble des hypothèses ciblées est vide, et l'algorithme essaye successivement de trouver des règles à ajouter à l'ensemble des hypothèses. Chaque règle ajoutée rendra certains exemples positifs vrais et tous les exemples négatifs faux. Lorsqu'une règle est trouvée, les exemples positifs qui sont couverts par la règle sont retirés de l'ensemble d'exemples positifs. De la même façon, le processus continu jusqu'à ce qu'aucune règle ne puisse être trouvée ou que l'hypothèse trouvée soit complète et consistante.

Dans la sous procédure de la recherche de la meilleure règle, la recherche commence avec la règle la plus générale de l'espace de recherche qui couvre tous les exemples. Durant la recherche, CN2 conserve un faisceau des règles candidates et la meilleure règle trouvée. À chaque étape dans la recherche du faisceau, tous les raffinements des règles dans le Beam sont considérés et évalués.

CN2 utilise une fonction heuristique d'évaluation de la qualité de la meilleure règle basée sur un estimateur de l'entropie. Cette mesure est utilisée pour diriger la recherche et pour déterminer la meilleure des règles trouvées dans le faisceau. L'espace de recherche peut être également réduit par élagage.

Une règle peut être élaguée s'il n'y a aucun raffinement qui puisse donner un meilleur résultat que la meilleure règle du moment. On peut arrêter le raffinement d'une règle lorsqu'elle ne couvre plus assez d'exemples positifs.

3.3.6.3 Le système FOIL

Quinlan a introduit le système FOIL en 1993, qui est une variante de l'algorithme de couverture séquentielle dans le cadre de la logique des prédicats. L'algorithme utilise des clauses de Horn dans lesquelles les prédicats ne peuvent pas avoir des fonctions parmi leurs arguments, ceci afin de réduire la complexité de l'espace de recherche.

Le but de FOIL est d'induire des règles jusqu'à ce que tous les exemples soient couverts. Chaque clause dans FOIL est générée par une approche du plus général au plus spécifique. En commençant par la clause la plus générale, FOIL spécialise la clause par construction de l'ensemble des littéraux candidats, puis il sélectionne le meilleur littéral parmi cet ensemble.

Le choix du meilleur littéral parmi l'ensemble des littéraux candidats est effectué par mesure du gain. Cette mesure évalue le nombre d'exemples positifs et négatifs couverts par la clause, avant et après l'ajout du littéral évalué. Dans FOIL, l'espace de recherche est borné, d'une part en limitant la complexité des clauses possibles à générer, d'autre part en ne considérant à chaque ajout, que les littéraux les plus discriminants.

3.4 Conclusion

Comme nous l'avons souligné, l'apprentissage de règles constitue une grande partie des travaux en apprentissage automatique. C'est un processus d'acquisition des connaissances par l'application d'inférences inductives sur des exemples afin de produire des énoncés généraux, des règles. Un tel processus comprend des opérations de généralisation, de spécialisation, de transformation, de correction et de raffinement des représentations des connaissances.

CHAPITRE IV

ENVIRONNEMENT EXPERIMENTAL

4.1 Introduction

L'objectif de ce chapitre est de présenter la partie application que nous avons développée pour concrétiser notre recherche. Notre travail consiste à construire des modèles de prédiction permettant de prédire les facteurs de qualité des logiciels tels que la maintenabilité et la réutilisabilité. Notre plateforme développée regroupe deux composants qui permettent l'extraction et la manipulation des connaissances.

Pour cela, on a recours aux fonctionnalités du logiciel Weka qui nous aident à faire de l'apprentissage automatique à partir d'une base de données et de produire des connaissances (règles), et aussi aux fonctionnalités du logiciel JRules qui nous permettent d'exécuter des règles sur de nouvelles données.

4.2 Description de Weka

Weka (Waikato Environment for Knowledge Analysis) est un environnement logiciel d'apprentissage développé par l'université Waikato en Nouvelle-Zélande. Il permet la manipulation et l'analyse de données. Son code source est écrit en Java et il est testé sur plusieurs plateformes telles que Linux et Windows. C'est un outil open source disponible sur le web.

Il contient la plupart des approches de l'apprentissage automatique, dont les arbres de décision, les réseaux bayésiens et les classificateurs basés sur les règles, etc. Les algorithmes peuvent être appliqués directement sur un ensemble de données ou appelés via un programme

Java. Sa propre interface graphique permet de charger un fichier de données, d'appliquer un algorithme d'apprentissage sur ces données et d'analyser les résultats.

Weka comporte plusieurs fonctionnalités :

- Explorateur (Explorer) : ce module regroupe tous les packages importants de Weka comme le prétraitement, les algorithmes d'apprentissage, le groupement ('clustering'), les associations, la sélection des attributs et la visualisation.
- Environnement expérimental ('Experimenter') : permet d'exécuter plusieurs algorithmes d'apprentissage en mode lot ('batch') et de comparer leurs résultats.
- Flot de connaissances ('Knowledge flow') : fournit les mêmes fonctionnalités que le composant 'Explorer'. Ces fonctionnalités sont représentées sous forme graphique en utilisant une interface de type 'drag-and-drop'.

Dans notre cas, nous s'intéressons juste au premier composant «Explorateur » qui nous aide à faire de l'apprentissage et à produire des connaissances.

4.2.1 Apprentissage avec Weka

L'interface principale du logiciel Weka est 'Explorer'. Cette interface sert principalement à construire des modèles de prédiction où la variable à prédire peut être nominale ou numérique. Cette interface possède plusieurs onglets qui donnent accès aux principaux composants de l'espace de travail.

Ce qui nous intéresse le plus dans le cadre de notre travail sont les deux onglets 'Preprocess' et 'Classify'. Le premier permet l'importation de données depuis une base de données sous le format 'arff'. Il permet aussi de filtrer ces données afin de transformer les types des attributs (ex : transformer un attribut numérique réel en un attribut discret).

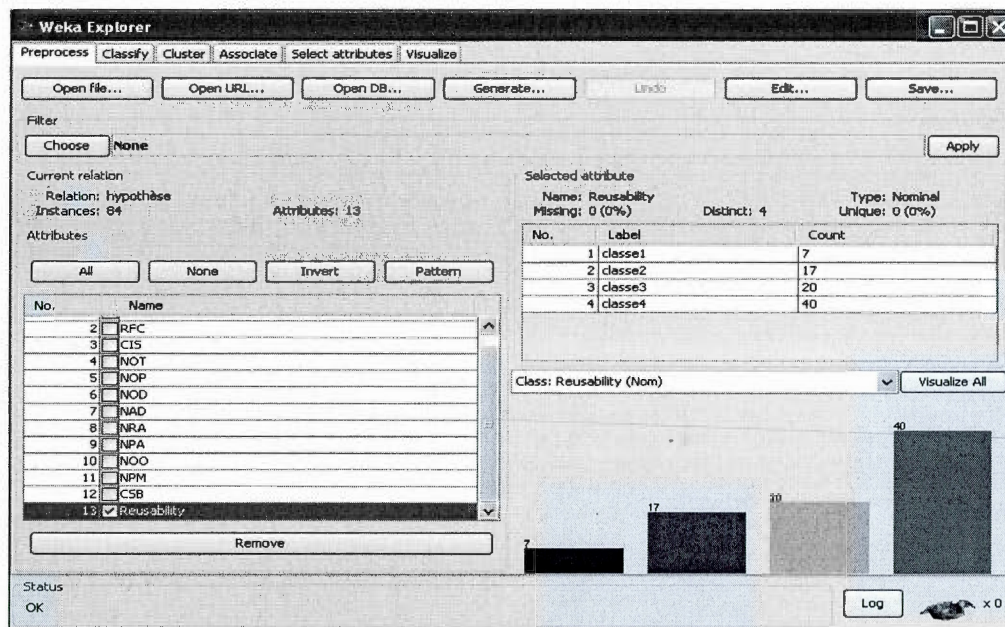

Figure 4.1 L'interface Explorer de Weka

Le deuxième onglet sert à appliquer des algorithmes d'apprentissage à l'ensemble des données, construire des modèles de prédiction soit sous la forme de règles (algorithmes à base de règles tel que Part), ou bien sous la forme d'arbres de décision (J48). Il permet aussi de visualiser les résultats de ces modèles.

Nous allons décrire dans ce qui suit le cheminement suivi dans la majorité des cas pratiques à savoir : la préparation des données, l'application d'un algorithme et l'analyse des résultats.

4.2.2 Préparation de données d'apprentissage (Preprocess)

L'objectif ultime consiste à réaliser l'apprentissage sur un ensemble de données et puis de valider le modèle sur un autre ensemble. Les données servant à la construction du modèle sont nommées données d'apprentissage. Quant aux données utilisées pour le test, elles sont appelées données de test et elles servent à calculer le taux d'erreurs du modèle

généré. Les ensembles de données d'apprentissage et de test doivent être complètement différents.

Dans notre cas, toutes les données disponibles représentent des mesures logicielles stockées dans des fichiers Excel. Dans Weka, nous ne pouvons charger que des fichiers du format 'arff' (format spécifique de Weka). C'est pour cette raison que, nos fichiers Excel doivent être convertis en des fichiers 'arff' pour être lu par 'Explorer'. Un fichier 'arff' a la forme suivante :

- Le nom de la relation qui est sous cette forme :

  @relation nom_des_données_d'Appretissage.

- La liste des noms des attributs avec leurs types.
- La liste des instances : chaque ligne représente une instance en termes de valeurs prises par chaque attribut.

Voici un exemple de fichier de données sous le format 'arff'.

hypoth1.arff - Bloc-notes

Fichier Edition Format ?

```
@relation hypothese1
@attribute Reusability {1,2,3,4}


@attribute DIT numeric
@attribute HIT numeric
@attribute NOA numeric
@attribute NOC numeric
@attribute NOD numeric
@attribute OIM numeric
@attribute NOP numeric

@data

4,0,0,0,0,0,7,0
4,0,0,0,0,0,7,0
2,0,0,0,0,0,12,2
4,0,0,0,0,0,15,0
4,0,2,0,1,2,31,12
4,0,0,0,0,0,19,0
3,1,0,1,0,0,14,5
3,2,0,2,0,0,14,5
4,1,0,1,0,0,17,1
4,0,0,0,0,0,11,0
2,2,0,2,0,0,20,10
4,0,0,0,0,0,15,0
4,0,0,0,0,0,12,0
3,1,1,1,2,2,12,5
3,2,1,2,2,2,10,4
4,1,0,1,0,0,21,1
```

Figure 4.2 Exemple d'un fichier arff

4.2.3 Application d'un algorithme d'apprentissage

Sous l'onglet Classify, les étapes à suivre sont:

- on choisit un algorithme d'apprentissage parmi ceux proposés par Weka, selon nos besoins.
- on choisi ensuite le type de test, il y'a ici 4 choix :
  1. utiliser tout l'ensemble d'apprentissage pour faire les tests (c'est l'option **Use training set**).
  2. sélectionner un autre ensemble de test (c'est l'option **Supplied test set**)
  3. la validation croisée (l'option **Cross-validation**) en choisissant le nombre de découpage appliqué sur l'ensemble d'apprentissage (dans notre travail pratique nous avons opté pour ce choix et nous avons mis le 'fold 'à 10). L'algorithme va apprendre 10 fois sur 9 parties et le modèle sera évalué sur la dixième partie restante. Les 10 évaluations sont alors combinées. Ce type de validation est recommandé lorsque l'ensemble des données n'est pas grand.
  4. partager l'ensemble de données en une partie pour l'apprentissage et une autre partie pour faire les tests (l'option **Percentage split**). (en général 2/3 pour apprentissage et 1/3 pour le test).
- lancer l'apprentissage et construire le modèle de prédiction.

Les résultats sont structurés en trois parties (figure 4.3) : la première partie représente les résultats sommaires, la deuxième donne les résultats par classes et la troisième représente une matrice de confusion. Les résultats sommaires donnent une statistique sur le nombre total d'instances correctement classifiées (les vrais positifs) et incorrectement classifiées, la valeur de Kappa, l'erreur absolue moyenne. Les résultats par classes nous fournissent le taux d'instances classifiées correctement et incorrectement via les taux TP ('true positif') et FP ('false positif'), la précision et d'autres informations statistiques. Pour la dernière partie, celle de la matrice de confusion, elle nous donne plus de précision concernant le nombre d'instances correctement et incorrectement classifiées correspondants aux taux TP et FP pour chaque classe.

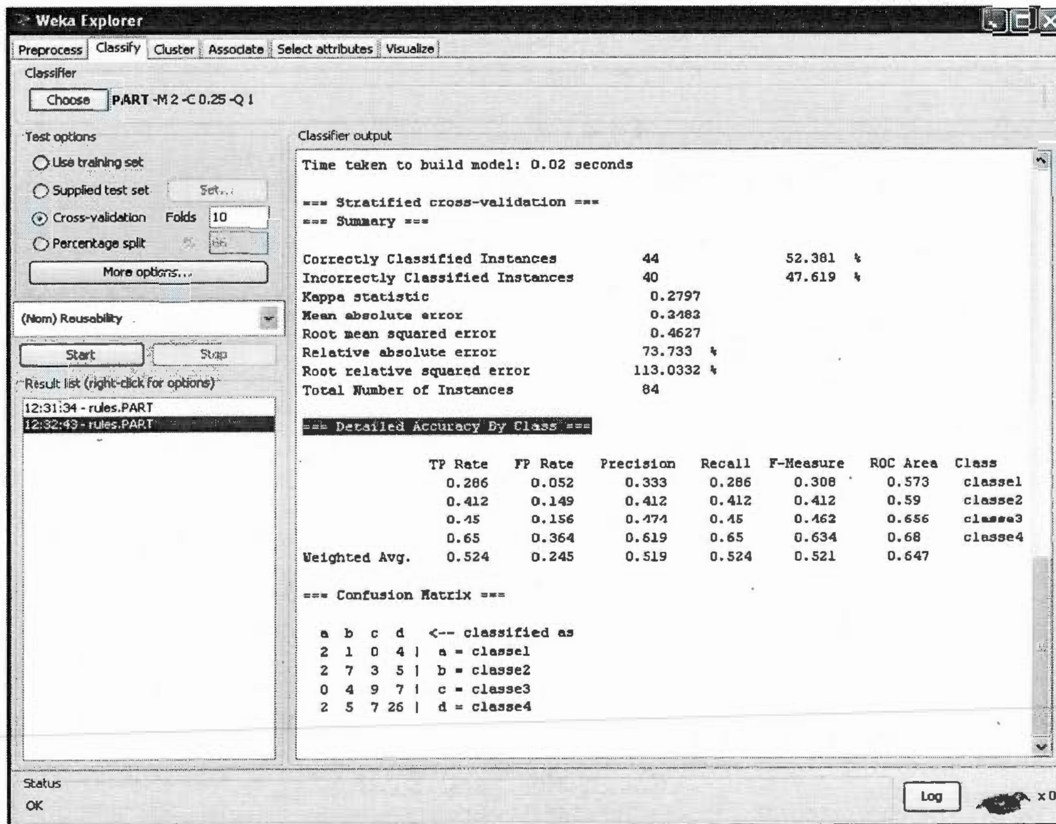

Figure 4.3 Résultats d'apprentissage

4.2.4 Les algorithmes utilisés dans notre approche

Dans cette section, nous décrivons les algorithmes utilisés dans notre application et qui sont disponibles dans le logiciel d'apprentissage Weka. Dans le cadre de notre travail, nous nous intéressons aux algorithmes d'apprentissage à base de règles. Nous avons sélectionné les algorithmes de classification les plus populaires et les plus adoptés à nos données qui sont : J48, Part, ConjunctiveRule et DecisionTable.

4.2.4.1 J48

L'algorithme J48 de Weka a été utilisé pour générer des arbres de décision. C'est une mise en œuvre de l'algorithme C4.5 de Quinlan en 1993 dans le cadre de l'apprentissage supervisé. Cet algorithme permet la construction d'un arbre de décision élagué ou non dont le

but de classifier des nouvelles instances avec un certain degré d'erreur tolérée. C4.5 commence par la phase d'expansion, il commence à construire l'arbre de décision en divisant récursivement l'ensemble d'apprentissage. Pour construire les sous arbres, C4.5 utilise une variable qui permet de calculer le taux de gain de l'information pour chacun des attributs possibles qui pourraient potentiellement être utilisés pour diviser les données. L'attribut qui a la plus grande valeur de gain est choisi comme racine d'un sous arbre. Cette méthode de construction des sous arbres est répétée plusieurs fois jusqu'à ce que l'arbre final classifie toutes les instances de l'ensemble d'apprentissage. Pour avoir un bon classificateur, l'arbre de décision doit être élagué. L'élagage de l'arbre de décision s'effectue en remplaçant un sous arbre entier par une feuille. Cette substitution a lieu si une règle de décision établit que le taux d'erreur attendu dans le sous arbre est supérieur que celui d'une simple feuille.

Voilà un exemple d'exécution de l'algorithme J48 sur quelques instances d'une base de données (la base de données est décrite dans le chapitre 5, section 5.5). Elle permet de classifier le niveau de réutilisation des composants logiciel.

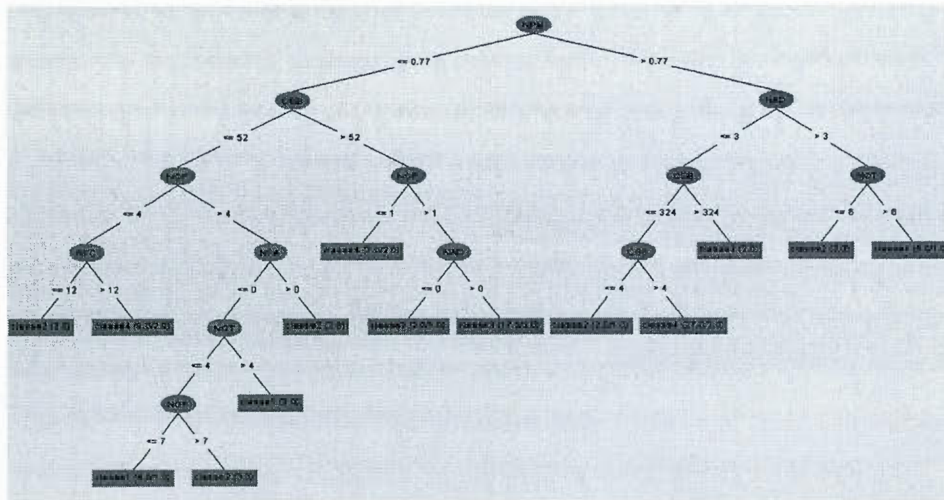

Figure 4.4 Résultats de classification de l'algorithme J48

4.2.4.2 Part

Cet algorithme permet de produire des règles par la génération itérative des arbres de décision partiel en combinant deux approches : Les avantages de C4.5 et la technique d'apprentissage automatique de règles « diviser pour régner ». Il fournit des résultats aussi précis que ceux de l'algorithme C4.5. PART permet d'éviter l'optimisation globale de C4.5 et fournit un ensemble de règles compactes et exactes (*Frank et al.*, 1998). Il adopte la stratégie de « diviser pour régner » avec laquelle il construit une règle, supprime les instances couvertes par cette dernière et continue à créer les règles récursivement jusqu'à ce qu'il ne reste aucune instance. En combinant les méthodologies, « diviser pour régner » avec les arbres de décision ajoute la flexibilité et la vitesse pour la génération des règles. Il est, en effet, inutile de construire un arbre de décision plein pour obtenir une règle simple. L'idée principale est de construire un arbre partiel de décision au lieu d'un arbre entièrement exploré.

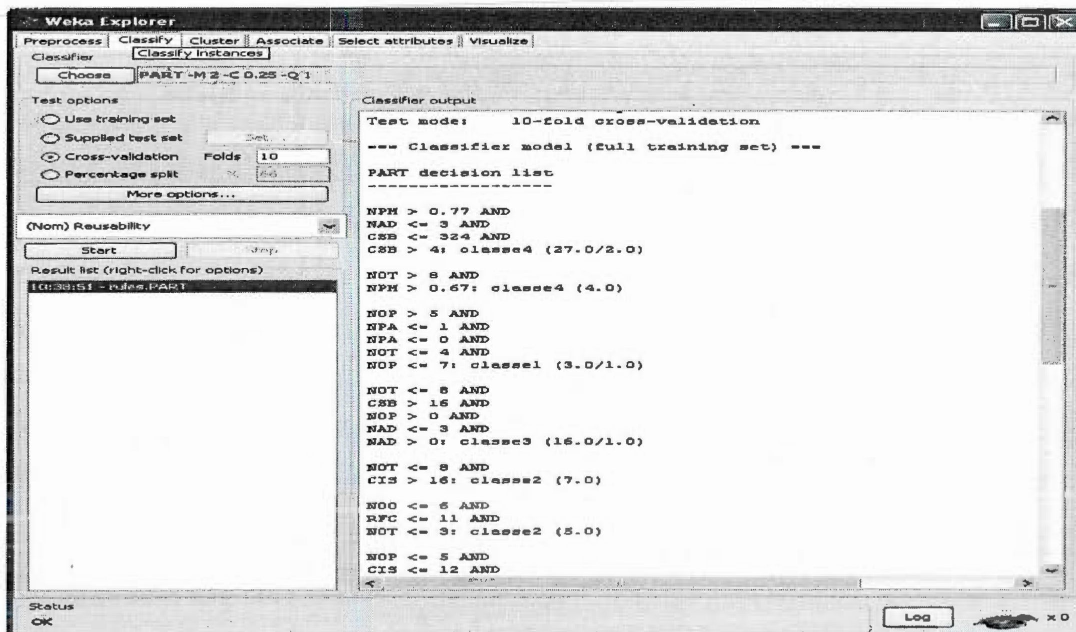

Figure 4.5 Résultats d'exécution de l'algorithme Part

4.2.4.3 ConjuntiveRule

Cette classe utilise une méthode d'apprentissage à base de règles conjonctives simples pour des classes numériques et nominales. Une règle se compose d'un ensemble de conditions liées ensemble par une conjonction AND et d'une conséquence. Chaque condition est une description sous la forme **attribute <op> value,** où **<op>** peut être soit un **<, <=, >, >=.** Dans le cas de la classification, la conséquence est la distribution des classes disponibles. Elle correspond à une moyenne pour la régression. Si une instance de test n'est pas couverte par cette règle alors la prédiction est la classe distributions/valeurs par défaut qui couvre les données non couvertes par aucune règle. Ce classificateur sélectionne un antécédent en calculant le gain de l'information de chaque antécédent et élague la règle générée en utilisant l'erreur réduite d'élagage (REP : Reduced Error Pruning) ou un pré-élagage simple basé sur le nombre d'antécédents. L'information d'un antécédent est la moyenne pondérée des entropies des données couvertes et non couvertes par la règle. Ci-bas, des résultats d'apprentissage de cet algorithme avec Weka.

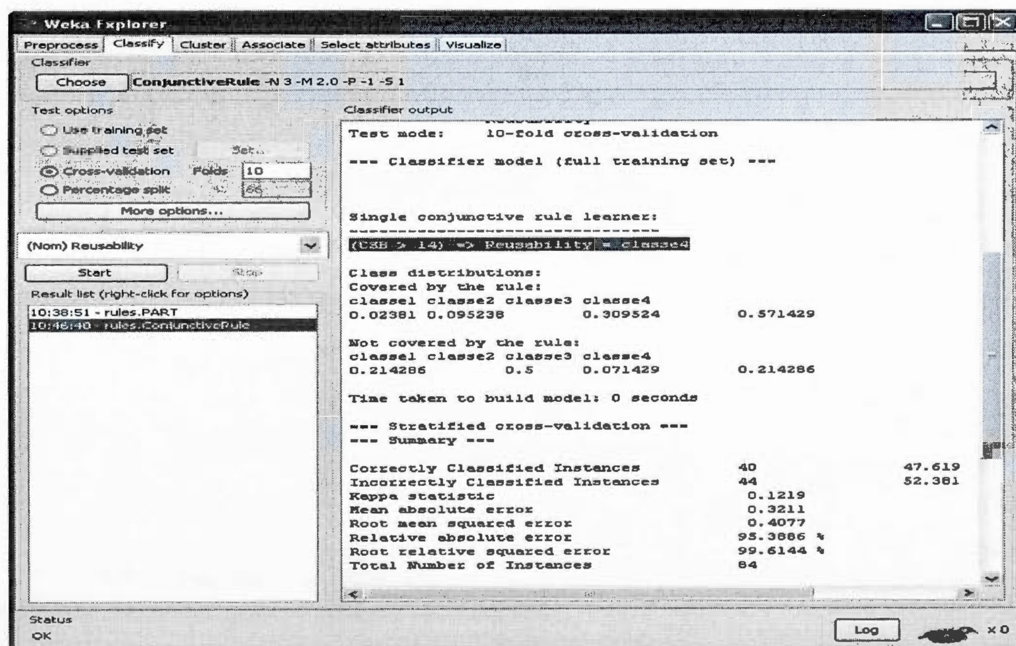

Figure 4.6 Résultats d'exécution de l'algorithme ConjuntiveRule

4.2.4.4 JRIP

L'algorithme JRIP de Weka correspond à la méthode Ripper de William W. Cohen (1995). Il permet de construire des règles itérativement pour couvrir les instances qui n'ont pas été couvertes auparavant. Les règles sont générées de la manière habituelle, mais de la classe la plus rare à la classe la plus fréquente. Ripper possède les mêmes avantages que C4.5, tout en étant bien plus efficace et surtout, il peut manipuler les données bruitées.

JRIP produit des règles indépendantes. Il intègre une première procédure de post-élagage afin de retirer les propositions inutiles, et une seconde procédure pour réduire le nombre de règles dans la base. Il en résulte souvent un classifieur plus compact par rapport aux autres algorithmes d'apprentissage.

Figure 4.7 Résultats d'exécution de l'algorithme JRIP

4.3 Description de JRules

Nous avons choisi d'implémenter nos règles en utilisant ILOG JRules. Ce dernier permet une exécution rapide même pour un nombre élevé de règles et cela comparativement à d'autres environnements à base de règles.

Dans JRules, une règle est écrite avec le langage ILR (ILOG Rule Langage) puis exécutée par le moteur de JRules. Le langage IRL a une syntaxe similaire à celle du langage Java et il permet d'utiliser les opérateurs et les collections Java dans l'expression des règles.

Une règle est composée d'une partie condition et d'une partie action. Elle a la forme when {*condition*} then {*action*}. Une condition peut porter sur un type, une relation, une annotation, etc. La partie *Action* décrit les actions à faire si la partie *Condition* est satisfaite. Dans notre cas, elle peut prédire un facteur de qualité d'une classe ou d'un composant d'un nouveau produit logiciel.

4.3.1 Le moteur d'inférence d'ILOG JRules

L'objectif d'un moteur d'inférence est de déterminer quelles sont les règles qui peuvent être appliquées. Il est composé de trois éléments essentiels :

- Une base de règles qu'on appelle le ruleset : elle contient les règles, les fonctions et les activités qui seront traitées par le moteur.
- Une mémoire de travail 'working memory' : elle contient les objets (des faits) qu'on insère. Les règles seront exécutées sur ces objets et leurs valeurs.
- Agenda : elle contient la liste des instances des règles en attente d'être exécutées sous la forme de paires <règle, n-uplet> ; tout n-uplet et règle sont tels que le n-uplet satisfait les conditions de la règle.

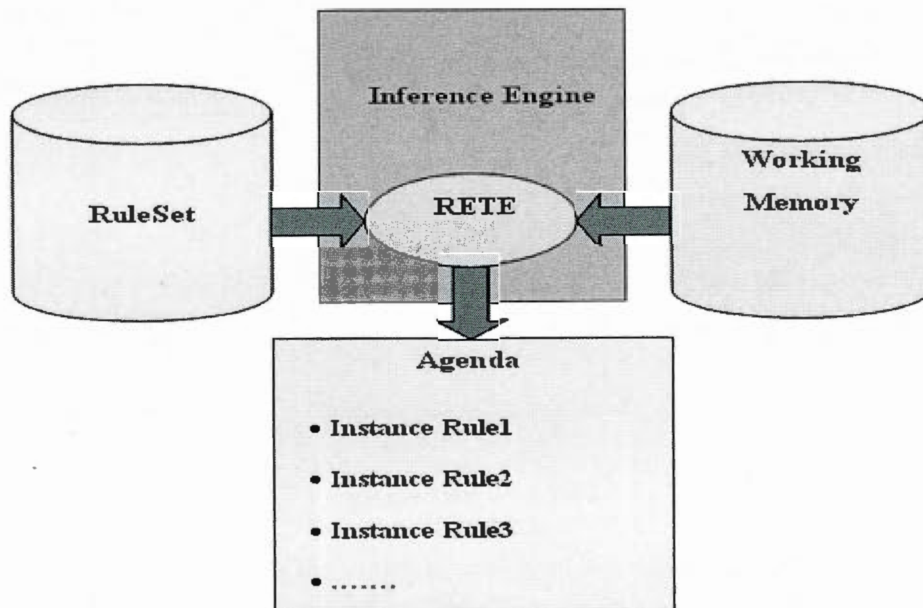

Figure 4.8 Moteur d'inférence de JRules

Le moteur d'inférence travaille en deux phases : la phase d'initialisation et la phase d'exécution.

L'initialisation du moteur consiste à :

- Compiler les règles afin de construire un réseau RETE servant à filtrer les règles éligibles en fonction des données. Une règle est dite éligible si ces conditions sont vérifiées à vrai sur un ensemble d'objets présents dans la mémoire de travail.

- Ajouter les objets au moteur.

La phase d'exécution se déroule en trois étapes :

- La première étape détermine les règles éligibles et les objets associés. L'ensemble « règle + objets associés » est appelé une « instance de règles ». Ces instances seront mises dans l'agenda.

- La deuxième étape consiste à choisir quelle règle sera exécutée dans l'agenda. Le choix de la règle dépend de sa priorité.

- La dernière étape consiste à exécuter la règle et mettre à jour les objets en apportant des modifications selon la base de faits.

Ainsi, le moteur JRules analyse les objets modifiées et réorganise la mémoire de travail. Cette mise à jour peut modifier l'agenda et d'autres règles peuvent s'y ajouter. Chaque cycle produit de nouvelles conclusions qui pourraient être considérées, à l'itération suivante, comme des faits sur lesquelles s'appliquent des règles. Ce processus peut se répéter pendant un certain nombre de cycles ou jusqu'à qu'aucune donnée ne vérifie les règles du système de production.

4.3.2 Règles JRules

La règle d'ILOG JRules a une structure simple, composée d'un entête, une partie pour les conditions et une partie pour l'action. La partie entête définit le nom de la règle, le paquet à laquelle la règle est attachée et sa priorité.

La partie des conditions utilise la structure orientée objet de Java pour effectuer le filtrage sur les instances de classe. Les conditions des règles sont également utilisées pour tester les valeurs de champ. Cela fournit un mécanisme de filtrage pour les objets. Lorsque la partie conditionnelle d'une règle est vérifiée, à savoir des objets valides ont été trouvés, la partie action de la règle peut être exécutée. Les actions peuvent varier du simple au complexe. La règle dans le langage de règles ILOG a la structure suivante:

```
rule myRule {
   packet = MyPacket;
   priority = high;
   when { conditions ... }
   then { actions ... }
};
```

Priorité d'une règle

La priorité de la règle détermine l'ordre dans lequel les règles sont exécutées dans une application. Elle détermine aussi la position d'une règle dans l'agenda. La priorité peut être un entier ou une expression entière. Si elle n'est pas spécifiée, la valeur 0 est utilisée. Il existe deux types de priorités, statiques ou dynamiques. Les priorités statiques sont des valeurs fixes, alors que les priorités dynamiques sont variables en fonction de valeurs déterminées par l'expression de la règle.

- ***Priorité statique*** : une priorité statique peut être utilisée pour modifier la séquence d'exécution d'une règle parmi d'autres différentes d'elle. Les priorités statiques sont des types entiers pour le langage Java. Plus le nombre est élevé, plus la priorité de la règle est forte. ILOG JRules fournit quatre valeurs de priorité prédéfini (minimum, bas, haut et maximum).
- ***Priorité dynamique*** : une priorité dynamique est utilisée pour modifier l'ordre d'exécution d'une même règle quand il existe différents instance de règles dans l'agenda. Les priorités dynamiques sont des expressions dont les valeurs dépendent d'une variable située dans la partie condition d'une règle.

4.3.3 Agenda

L'agenda est un endroit dans le moteur d'inférence où on peut mettre en file d'attente les instances de règles à exécuter. L'agenda fonctionne selon une stratégie bien définie : lorsque les conditions d'une règle donnée sont satisfaites et qu'un fait est déclaré dans la mémoire de travail, la règle est placée dans l'agenda et elle sera exécutée selon sa priorité. Les critères d'ordonnancement des règles dans l'agenda sont :

- réfraction : une règle déjà exécutée ne peut être réinsérée dans l'agenda que s'il y a un nouveau fait qui se produit;

- priorité : c'est le deuxième critère qui permet de décider la position que peut prendre une règle dans l'agenda;

- recency : si deux instances de règles ont la même priorité, la règle associé à l'objet le plus récemment modifié ou inséré, sera exécutée en premier;
- ordre lexicographique : deux règles ont la même priorité et la même regency, l'ordre d'exécution va être déterminé selon l'ordre lexicographique des noms de règles.

La priorité, recency et l'ordre lexicographique sont utilisés pour résoudre le conflit quand plusieurs instances de règles sont sur le point de s'exécuter en même temps.

4.3.4 Algorithme RETE

L'algorithme RETE (Le mot RETE signifie réseau en italien) permet de « filtrer » les objets de la mémoire de travail pour déterminer les règles éligibles et les instances de règles d'une manière optimisée. Comme son nom l'indique, le principe est d'utiliser un réseau prenant comme source les objets de la mémoire de travail pour établir des feuilles représentant les règles qui seront ajoutées à l'agenda. Le réseau RETE est formé de 5 types de nœuds : la racine, les nœuds type, les nœuds alpha, le nœud bêta et les feuilles. Le nœud racine est le point d'entrée de tous les objets. Les nœuds intermédiaires (type, alpha et bêta) représentent des conditions des règles. Les nœuds alpha servent à faire des tests de filtrage, ils représentent les littéraux de la partie condition d'une règle. Les nœuds beta sont des nœuds de jointures. Ces derniers reçoivent comme entrée plusieurs faits et copient en sortant le résultat de l'opération. Tous les objets, passent alors par ce réseau est subissent des tests à chaque nœud, Si ces objets arrivent à une feuille du réseau RETE, la règle correspondante est ajoutée à l'agenda (documentation JRules).

Si on exécute chaque règle sur chaque fait dans l'ordre, Le moteur de règles sera alors un facteur de ralentissement et non d'amélioration des performances. La sophistication de l'algorithme RETE permet de gagner du temps grâce à des mécanismes tel que le partage de

conditions, où une condition partagée par plusieurs règles peut être évaluée une seule fois pour toutes les règles.

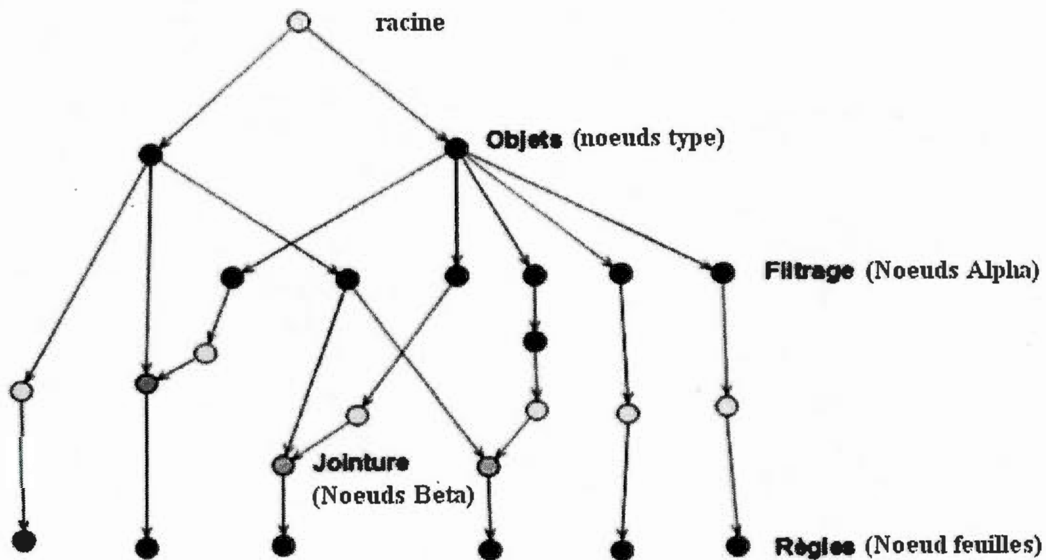

Figure 4.9 Réseau RETE

4.4 Conclusion

À travers ce chapitre, nous avons présenté les deux environnements expérimentaux Weka et JRules que nous utiliserons lors de développement de notre outil. Weka nous permet de faire de l'apprentissage automatique et de créer des modèles de prédiction, tandis que JRules permet d'exécuter ces modèles sur de nouveaux objets.

CHAPITRE V

INPLÉMENTATION DU PROTOTYPE ET EXPÉRIMENTATION

5.1 Introduction

L'objectif de ce chapitre est de formaliser et représenter des modèles qui prédisent des facteurs de qualité tels que le coût d'une maintenance corrective, la réutilisabilité et la prédisposition aux fautes pour des nouveaux logiciels. Nous construirons des modèles de prédiction par l'application de techniques d'apprentissage automatique sur des mesures d'attributs internes (couplage, cohésion, taille, complexité, etc.) qui sont collectées sur des applications réelles.

Avant l'implémentation de notre prototype, nous avons conçu son architecture, défini ses modules, identifier ses utilisateurs potentiels et choisit les technologies adéquates et les outils efficaces à utiliser.

5.2 Modules du prototype

Le prototype de notre système d'aide à la décision pour la conception en génie logiciel est constitué de quatre principaux modules qui sont les suivants :

- Préparation des données : conversion de données Excel en format arff manipulable par le logiciel Weka ; création de classes Java à partir de fichiers arff pour les utiliser avec le logiciel JRules afin de formaliser et représenter des modèles prédictifs.

- Apprentissage automatique : apprentissage qui peut se faire en mode débutant, expert et mène à la génération des modèles de prédiction.

- Construction d'une base de règles à partir des modèles générés par les algorithmes d'apprentissage (des fichiers '.ilr' manipulables par le logiciel JRules)

- Prédiction et prise de décision en exécutant les règles (.ilr) sur de nouvelles données extraites du code source de nouvelles applications

Les sections suivantes expliquent en détail les composants et les modules du prototype.

5.3 Éclipse et Java comme infrastructure d'implémentation

5.3.1 Java

Java est un langage de programmation orienté objet. Il est conçu pour que les programmes qui l'utilisent soient fiables sous différents aspects. Java a la particularité d'être portable sur plusieurs systèmes d'exploitation tels que Windows ou Linux. L'interpréteur Java peut exécuter les bytecode directement sur n'importe quelle machine sur laquelle il a été porté. Sa bibliothèque d'exécution est indépendante de la plateforme.

5.3.2 Éclipse

Eclipse est un environnement de développement intégré qui est utilisé pour créer diverses applications. C'est un logiciel libre «open source ». Il fournit un Framework basé sur les plug-ins; Le Framework facilite la création, l'intégration et l'utilisation d'outils logiciels. Sous Eclipse, nous avons développé notre plateforme intégrant les deux logiciels Weka et JRules. Nous avons pu aussi concevoir et implémenter les différentes interfaces graphiques à l'aide de l'éditeur visuel d'Eclipse. Dû au fait qu'Eclipse est basé sur une architecture ouverte et un modèle de plug-ins, nous le considérons comme le meilleur outil pour implémenter notre prototype.

5.4 Diagramme de cas d'utilisation

Le diagramme des cas d'utilisation décrit ci-dessous nous donne une vision globale du comportement fonctionnel de notre prototype. Nous allons décrire toutes les fonctionnalités ainsi que les acteurs (les utilisateurs) de notre système.

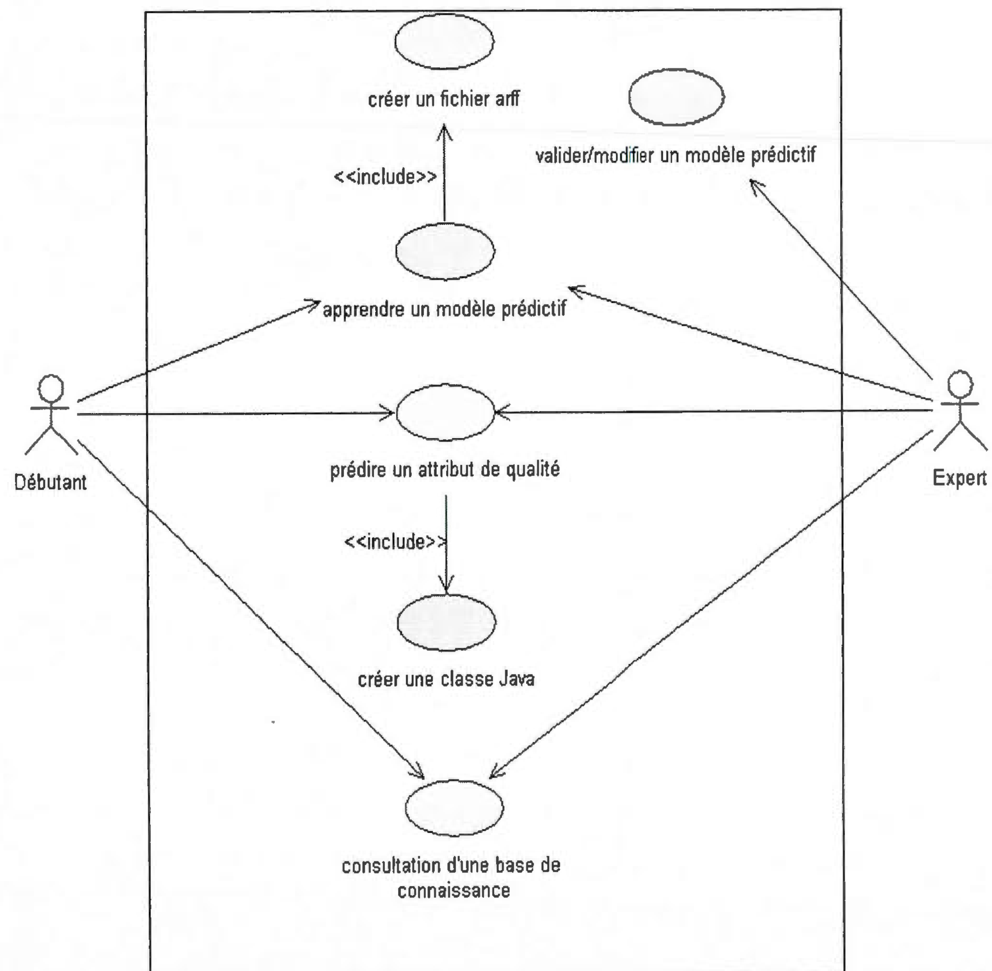

Figure 5.1 Diagramme de cas d'utilisation de système d'aide à la décision

5.4.1 Les acteurs

Les utilisateurs de notre système d'aide à la décision sont regroupés en deux catégories : les débutants qui sont des utilisateurs novices et inexpérimentés dans le domaine du génie logiciel, et nous avons aussi les experts qui ont la capacité, du fait de leur expérience, d'exprimer des jugements pertinents dans certaines situations.

5.4.2 Les différents cas d'utilisation

Notre prototype permet de faire deux tâches principales : l'apprentissage et la prédiction. Pour faire de l'apprentissage, nous utiliserons les fonctionnalités offertes par le logiciel Weka, et pour prédire un facteur de qualité, nous nous baserons sur le logiciel JRules. Les différents cas d'utilisation que nous avons développés sont offerts à nos deux types d'acteurs le débutant et l'expert.

Un débutant peut faires les tâches suivantes :

- Apprendre un modèle prédictif (base de connaissances) qui nécessite la création d'un fichier arff manipulable par le logiciel Weka à partir d'un fichier de données Excel.
- Consulter une base de connaissances (modèle de prédiction).
- Estimer/prédire un attribut de qualité (exécuter un modèle) qui nécessite la création d'une classe Java qui correspond aux objets à prédire.

Un expert peut lui aussi solliciter toutes ces tâches avec la possibilité de créer et modifier les bases de connaissance (ajouter, supprimer ou modifier une condition dans une règle, etc.) afin de valider un modèle. Toutes les fonctionnalités de notre outil sont résumées dans le diagramme de cas d'utilisation suivant :

Le logiciel Weka, comme vu dans le chapitre 4, ne manipule que des fichiers au format arff. Quant à JRules, lors de l'exécution d'une base de règles, il analyse les objets présents

dans sa mémoire de travail. Ces objets sont des instances de classes Java ; on doit donc créer nos fichier arff et nos classes Java avant l'exécution de tels scénarios.

5.5 Expérimentation

Nous présentons dans ce chapitre, les différentes fonctionnalités de notre outil commençant par la préparation des données, l'apprentissage et enfin la prédiction. L'interface principale de notre prototype contient une barre de menus qui possède quatre menus : Java, Weka, JRules et Decision engine.

Dans le menu Java, nous avons un item qui s'appelle 'Classe java' qui nous permet de créer une classe java.

Dans le menu Weka, nous avons deux items qui sont : 'Fichier arff' pour créer un fichier 'arff' et 'Apprentissage' pour faire l'apprentissage en utilisant les fonctionnalités de Weka.

Dans le menu JRules, nous avons aussi deux items : « Débutant » et « expert ». Le premier permet à un débutant en conception en génie logiciel de consulter une base de connaissances et le second permet à un expert de consulter ou de modifier une base de connaissances.

Le dernier menu est celui du 'decision engine'. Il contient lui aussi deux items : le premier nous sert à importer de nouveaux objets d'une application à analyser (par exemple des classes pour lesquelles nous voulons faire une prédiction). Le second nous permet donc d'exécuter les modèles de prédiction à l'aide du moteur JRules.

Le but de nos expérimentations est de prédire un facteur de qualité. Pour la démonstration que nous allons faire,nous avons choisi la réutilisabilité vu l'importance de ce facteur pour l'approche orientée-objet et en en raison aussi de la disponibilité d'une base de données.

Pour prédire la réutilisabilité de nouveaux composants logiciels, nous nous sommes basés sur le travail de Mao (1998). Nous essayerons de vérifier certaines hypothèses sur la réutilisabilité. Ces hypothèses concernent l'impact de trois attributs internes de qualité, l'héritage, le couplage et la complexité, sur la réutilisabilité de classes dans les applications orientées-objets.

Les hypothèses que nous pouvons vérifier dans ce contexte sont :

- ***Hypothèse-HE-REUT*** : dans quelle mesure la position d'une classe dans la hiérarchie d'héritage peut-elle affecter la réutilisabilité?
- ***Hypothèse-COUP-REUT*** : comment le couplage entre une classe et son environnement peut affecter la réutilisabilité de cette classe?
- ***Hypothèse-COMP-REUT*** : comment la taille et la complexité d'un composant logiciel ont un impact sur la réutilisabilité?

Les données, sur lesquelles nous avons travaillé, ont été collectées du système multi-agents LALO (Language d'Agents Logiciel Objets), système développé et maintenu depuis 1993 au CRIM (Centre de Recherche Informatique de Montréal). Ce système est composé de 84 classes en C++ et dont la taille est de 47 K lignes de code. Deux outils ont été utilisés pour extraire les mesures à savoir QMOOD et GEN++. Les mesures sont dérivées de l'analyse statique du système LALO (seules les classes développées sont considérées).
Pour définir le niveau de réutilisabilité d'un composant logiciel, une classification a été proposée par les développeurs spécialistes dans le domaine des systèmes multiagents :

- Totalement réutilisable : formé des composants génériques à un certain domaine (dans le cadre de ce travail, le domaine est le développement de systèmes multiagents) ; cette classe est désignée par ***classe1***.
- Réutilisable avec un minimum de travail : moins de 25% du code source nécessite d'être modifié pour être réutilisé ; cette classe est identifiée par ***classe2***.

- Réutilisable avec de grands changements : plus de 25% de code source doit subir des changements pour que le composant logiciel soit réutilisable ; nommée ***classe3***.
- Non réutilisable : composants ne pouvant pas être réutilisés. Ils sont spécifiques à l'application ; désignée par ***classe4***.

Nous avons choisi de vérifier l'hypothèse suivante : ***Hypothèse-COMP-REUT***.
La liste des mesures pour vérifier l'influence de la complexité sur le facteur réutilisabilité est la suivante :

- Response For a Class (RFC)
- Number Of local Methods (NOM)
- Class Interface Size (CIS)
- Number Of Parameters Per Method (NPM)
- Number Of Abstract Data Types (NAD)
- Number of Reference Attributes (NRA)
- Number of Public Attributes (NPA)
- Class Size in Bytes (CSB)
- Number of Trivial Methods (NOT)
- Number Of Parents (NOP)
- Number Of Descendents (NOD)
- Number of Overloaded Operators (NOO)

La définition des métriques ci-dessus et la signification de leurs valeurs possibles ainsi que leurs interprétations ont été présentées au chapitre 1.

5.5.1 Création d'un fichier arff à partir d'un fichier Excel

Toutes les données que nous avons sont sur des fichiers Excel. À l'aide de notre outil, l'utilisateur, soit un expert, soit un débutant, peut convertir un fichier Excel en un fichier arff. Les transformations consistent à :

- Ajouter en entête du fichier les informations nécessaires pour un fichier 'arff'

1. définir des données d'apprentissage par une relation sous la forme de :

   @relation nom-des-données-d'apprentissage.

   Exemple : *@relation* PredictedClasse.

2. définir les différents attributs (ici, les métriques dans l'ordre d'apparition) avec leurs types,

   Exemple : *@attribute* LCOM 1 real.

3. rajouter la variable à prédire avec son type.
4. signifier le début des données (vecteurs) par le mot réservé *@data*

- Écrire le contenu des cellules situées dans la même ligne dans le fichier Excel, sur une même ligne dans le fichier arff en séparant les valeurs par des virgules.

- Sauvegarder le fichier transformé des données sous l'extension '.arff'.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	NOM	RFC	CIS	NOT	NOP	NOD	NAD	NRA	NPA	NOO	NPM	CSB	Reusability
2	7	12	7	2	0	3	3	0	0	1	0,71	88	classe4
3	7	13	7	1	0	1	1	0	0	1	0,57	76	classe4
4	12	54	12	1	2	2	2	1	0	2	0,83	80	classe2
5	15	26	15	2	0	2	1	1	0	4	0,87	[illegible]	classe4
6	31	92	16	17	12	13	3	6	0	1	0,71	268	classe4
7	19	106	19	18	0	4	4	0	0	3	1,26	16E	classe4
8	14	24	14	6	5	3	3	0	1	4	0,71	9E	classe3
9	14	22	14	8	5	4	4	0	1	6	0,71	264	classe3
10	17	27	17	6	1	3	3	1	0	4	2,18	16	classe4
11	11	29	11	0	0	2	1	0	0	1	1,45	80	classe4
12	20	51	20	4	10	3	2	2	1	2	0,75	8	classe2
13	15	64	15	2	0	3	2	0	0	1	2,27	156	classe4
14	12	29	12	4	0	3	2	0	0	2	0,67	204	classe4
15	12	11	12	5	5	2	2	0	0	3	0,58	56	classe3
16	10	11	10	5	4	2	2	0	0	3	0,6	132	classe3
17	21	23	21	2	1	3	3	2	0	2	2,52	148	classe4

Figure 5.2 Extrait d'un fichier Excel d'une base de données

Voici un exemple de fichier de données sous le format '.arff' tel que créé par notre outil et qui correspond à la 1ère hypothèse ***Hypothèse-COMP-REUT***.

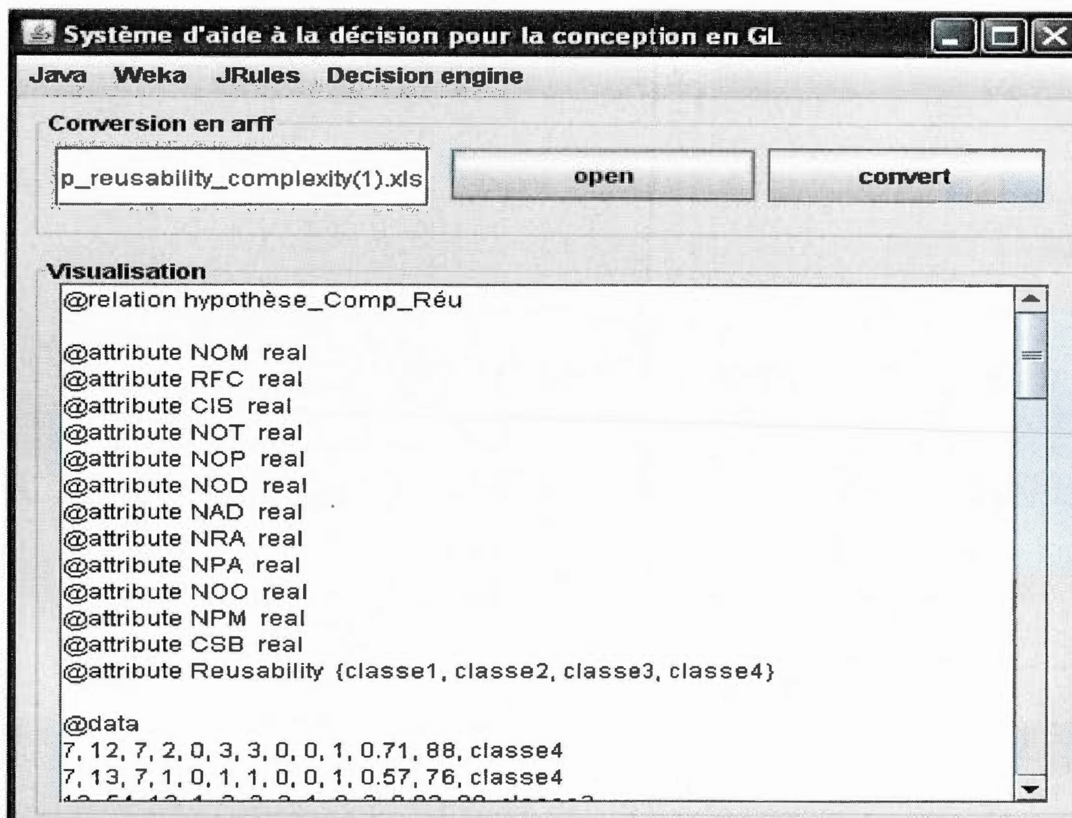

Figure 5.3 Résultat de conversion du fichier Excel en un fichier 'arff'

5.5.2 Création d'une classe Java qui correspond à l'hypothèse Hypothèse-COMP_REUT

Le moteur ILOG JRules exécute une base de règles sur des objets. Ces objets sont des nouveaux composants dont nous désirons prédire le niveau de réutilisabilité dans le futur.

La première étape pour pouvoir manipuler ces objets (composants logiciels) est de créer une classe Java qui sert à décrire ces objets, c'est-à-dire, définir la structure d'un objet. Cette définition avec Java se fait de la manière suivante : déclaration des données membres et déclaration des méthodes. Les données membres permettent de conserver des informations relatives à la classe, tandis que les méthodes représentent les traitements qu'il est possible de réaliser avec les objets instanciés de la classe.

L'ensemble de données d'apprentissage est formé de mesures de 12 métriques liées à la complexité et la taille d'un composant logiciel. La classe que nous allons créer à la structure suivante :

- Le nom de la classe est « Composant ».
- Les attributs de la classe sont les métriques NOM, RFC, CIS, NOT, NOP, NOD, NAD, NRA, NPA, NOO, NPM, CSB et la variable à prédire 'Reusability'.
- Les méthodes sont le constructeur de la classe et les accesseurs.

Nous exploitons le fichier 'arff' pour créer la classe correspondante à l'hypothèse *Hypothèse-COMP-REUT*. L'utilisateur (débutant ou expert) commence par importer le fichier arff pour prendre les noms des attributs, il saisit ensuite le nom du package où le programme s'exécute. Il saisit aussi le nom de la classe et a la possibilité d'ajouter ou de supprimer des attributs. Enfin, il appuie sur le bouton 'Create Java class'. Le résultat de l'exécution est dans la figure suivante :

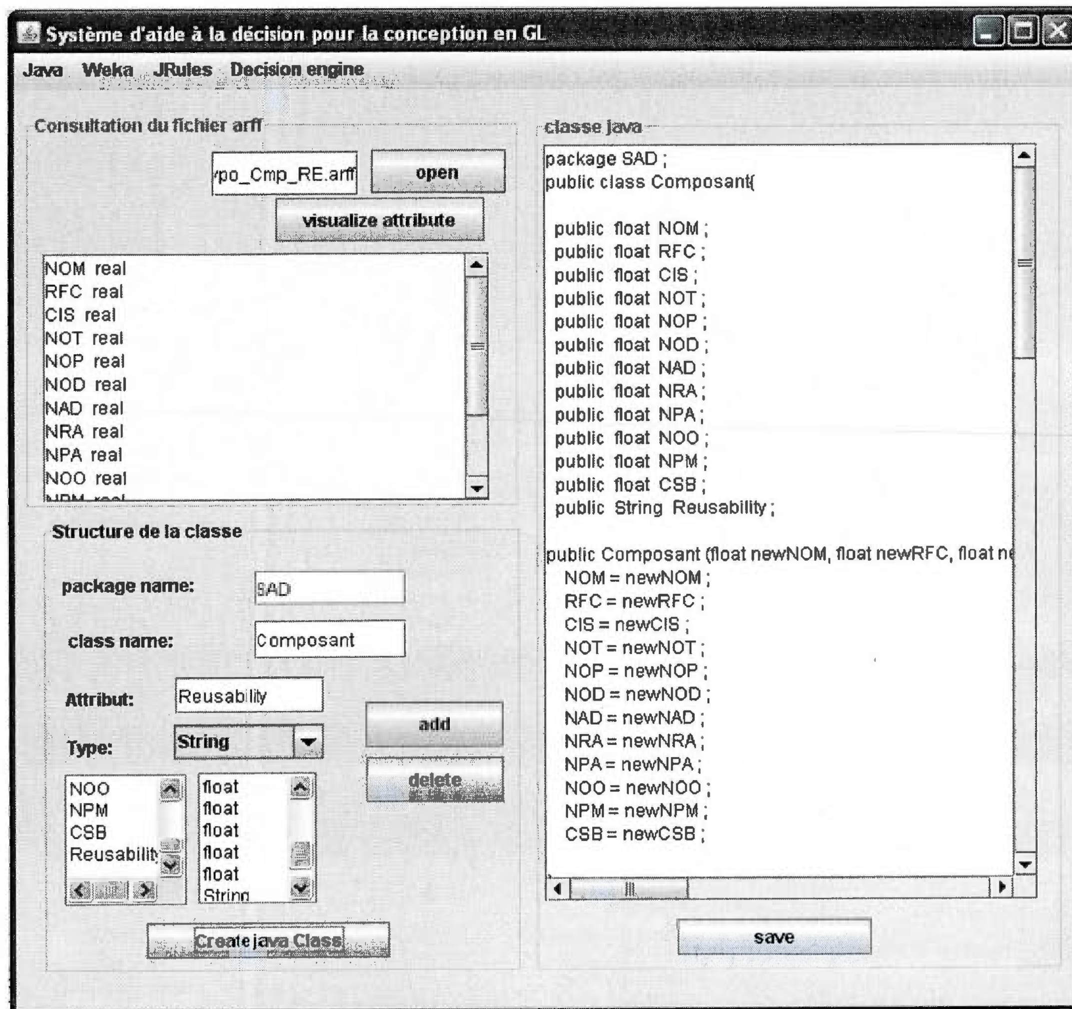

Figure 5.4 Création d'une classe Java des objets à prédire

5.5.3 Apprentissage automatique

Une des fonctionnalités les plus importantes de notre outil est l'apprentissage; il nous permet de construire les modèles prédictifs.

Avant d'aborder la présentation des résultats, nous définissons le mode de test utilisé lors de l'apprentissage et les critères d'évaluation des modèles. L'application des algorithmes

d'apprentissage sur les bases de données est accomplie en mode validation croisée stratifiée 10-fold. Nous avons choisi ce mode pour deux raisons :

- Ce mode d'expérimentation est adéquat pour les bases de données de petite taille puisque la base de données que nous exploitons contient seulement 83 objets.
- La validation croisée permet de générer des résultats pertinents.

L'hypothèse à vérifier est la suivante : « est-ce que la taille et la complexité d'un composant logiciel affecte d'une certaine manière sa réutilisabilité ? ».

Notre prototype permet de faire la tache de l'apprentissage avec trois algorithmes qui sont déjà décrits dans le chapitre 4: Part, ConjunctiveRule et JRIP.

Nous présentons, dans cette section, les résultats de l'application de l'algorithme Part. Cet algorithme permet de produire des règles par la génération itérative d'arbres de décision. Les résultats sont des règles de la forme (condition et action) accompagnées d'un couple de nombres écrits entre parenthèses.

Le premier nombre entre parenthèse représente le nombre d'instances correctement classifiées ***Correctly Classified Instances*** et le second représente le nombre d'instances incorrectement classifiées ***Incorrectly Classified Instances.*** S'il y'a un seul nombre, il s'agit, alors, du nombre d'instances correctement classifiées et qu'il n'y a pas d'instances mal classifiées avec cette règle.

Le modèle de classification généré par l'algorithme Part consiste en 11 règles, 10 règles explicites et 1 règle par défaut. Ces règles sont :

NPM > 0.77 AND NAD <= 3 AND CSB <= 324 AND CSB > 4: classe4 (27.0/2.0)

NOT > 8 AND NPM > 0.67: classe4 (4.0)

NOP > 5 AND NPA <= 1 AND NPA <= 0 AND NOT <= 4 AND NOP <= 7: classe1 (3.0/1.0)

NOT <= 8 AND CSB > 16 AND NOP > 0 AND NAD <= 3 AND NAD > 0: classe3 (16.0/1.0)

NOT <= 8 AND CIS > 16: classe2 (7.0)

NOO <= 6 AND RFC <= 11 AND NOT <= 3: classe2 (5.0)

NOP <= 5 AND CIS <= 12 AND NRA <= 0: classe4 (9.0/1.0)

NAD <= 2 AND NOM > 10: classe1 (5.0)

NRA > 1: classe4 (4.0/1.0)

NPM <= 0.7: classe2 (2.0)

: classe3 (2.0)

Une fois le modèle de classification induit par Part, l'étape suivante consiste à évaluer ce modèle afin de déterminer son efficacité prédictive. Un modèle possédant une bonne efficacité prédictive implique que les métriques utilisées sont pertinentes pour identifier les classes ou les composantes logicielles susceptibles d'être réutilisés dans le futur. Pour connaitre la cohérence et la complétude d'un modèle, il faut calculer le pourcentage d'efficacité de chacune de ces règles.

Au sein d'un tel scénario, un expert en génie logiciel qui a une bonne expérience peut soumettre des propositions afin de modifier la structure d'une règle.

Voilà le résultat de notre outil lors de l'apprentissage avec l'algorithme Part.

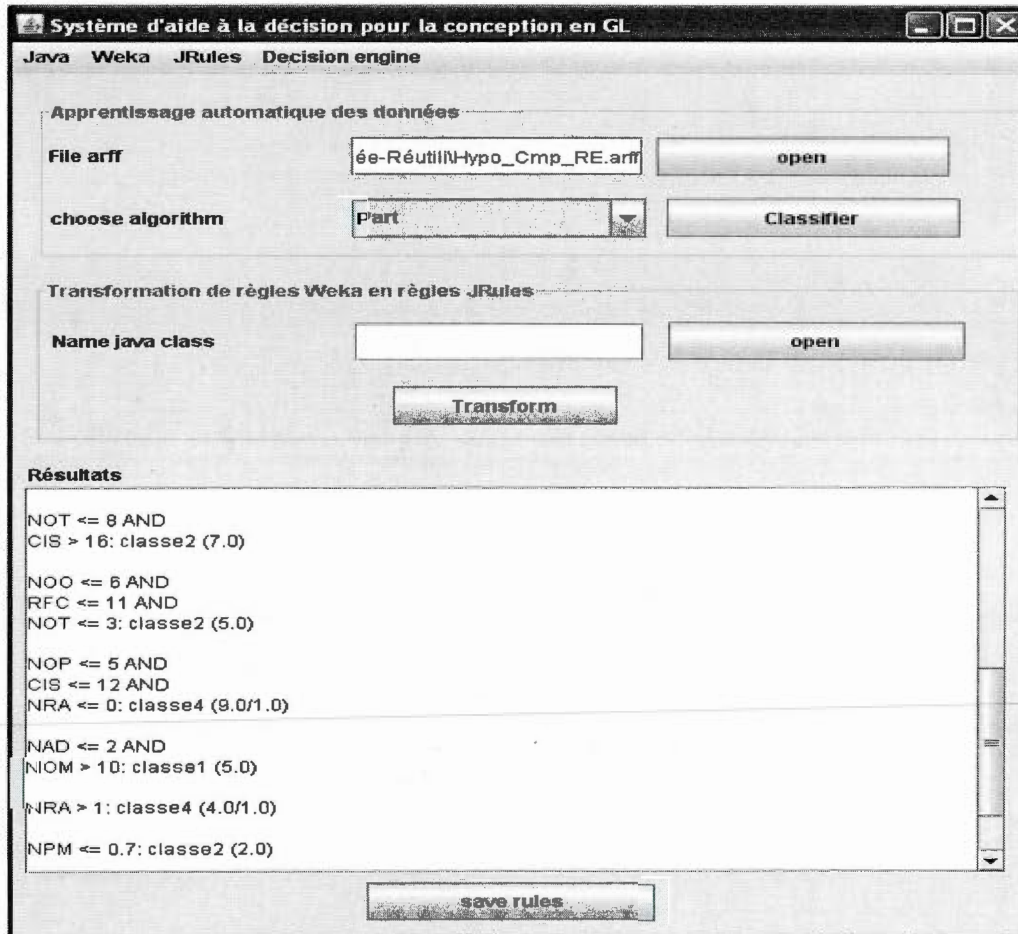

Figure 5.5 Résultat d'apprentissage avec l'algorithme Part

5.5.4 Construction d'une base de règle (fichier .ilr)

Nous avons choisi d'implémenter les règles en utilisant ILOG JRules. Ce dernier permet une exécution rapide même pour un nombre élevé de règles. Ainsi, les règles produites par les algorithmes d'apprentissage devront être transformé en règles exécutables avec JRules. Comme c'est mentionné dans le chapitre précédent, les règles de JRules ont une syntaxe très semblable du code Java. Nous commencons d'abord par donner à chacune des règles une priorité, et cette dernière sera calculée de la façon suivante : les nombres écrits entre parenthèse dans une règle produite par apprentissage et qui représentent le nombre

d'instances correctement classifiées et les instances incorrectement classifiées, sont transformés en une priorité pour la règle. Notons :

NICC : Nombre d'Instances Correctement classifiées

NIIC : Nombre d'Instances Incorrectement Classifiées

Donc : **Priority (%) = (NICC / (NICC+ NIIC)) *100.**

Dans JRules, la priorité d'une règle doit être un entier. Puisque cette formule nous donne des réels (des pourcentages), alors nous devons transformer ces résultats en des entiers et attribuer un nombre entier comme priorité pour chaque règle. La règle qui a le pourcentage le plus élevé doit avoir la priorité la plus grande et les règles qui ont le même pourcentage auront la même priorité. Si par exemple, nous avons 4 règles avec des pourcentages différentes (100 %,100%, 89.99 %, 50 %), alors la première et la deuxième règle auront la plus grande priorité qui est 3, la troisième aura la priorité 2 et la dernière aura la priorité 1. Lors de l'exécution, le moteur JRules commence par exécuter la règle qui a la plus grande priorité. Voici un exemple de transformation d'une règle Weka en une règle JRules :

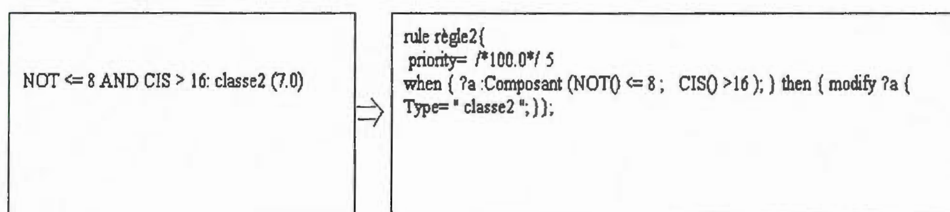

Figure 5.6 Exemple de règles JRules

L'interface de l'apprentissage nous permet de faire cette transformation automatique. Comme les règles JRules manipulent des objets, nous avons donc besoin d'importer la classe Java qui correspond à ces objets. Nous appuyons, alors, sur le bouton 'Transform' pour visualiser ces transformations. Le résultat va être des règles telles que codées en JRules. Elles seront aussi organisées selon leurs priorités. La première aura la plus haute priorité et la dernière aura la plus petite priorité. La règle aura 4 parties : Le nom, la priorité, les conditions et l'action. La

dernière étape est de sauvegarder notre base de règles dans un fichier d'extension '.ilr', le format spécial de JRules.

Figure 5.7 Création d'un fichier .ilr

5.5.5 Consultation ou modification d'une base de règles

Nous décrivons dans cette section deux autres fonctionnalités offertes par notre outil que nous trouvons dans le menu 'JRules'. Ce Menu contient deux items, le premier concerne un nouvel utilisateur que nous avons appelé débutant, et le second concerne un expert dans le domaine. Un débutant ne peut que consulter une base de connaissances (règles). Par contre

un expert peut évidemment proposer de nouvelles connaissances ou les modifier. La seule différence entre les deux interfaces est que celle du débutant n'est pas éditable, par contre, celle de l'expert est éditable.

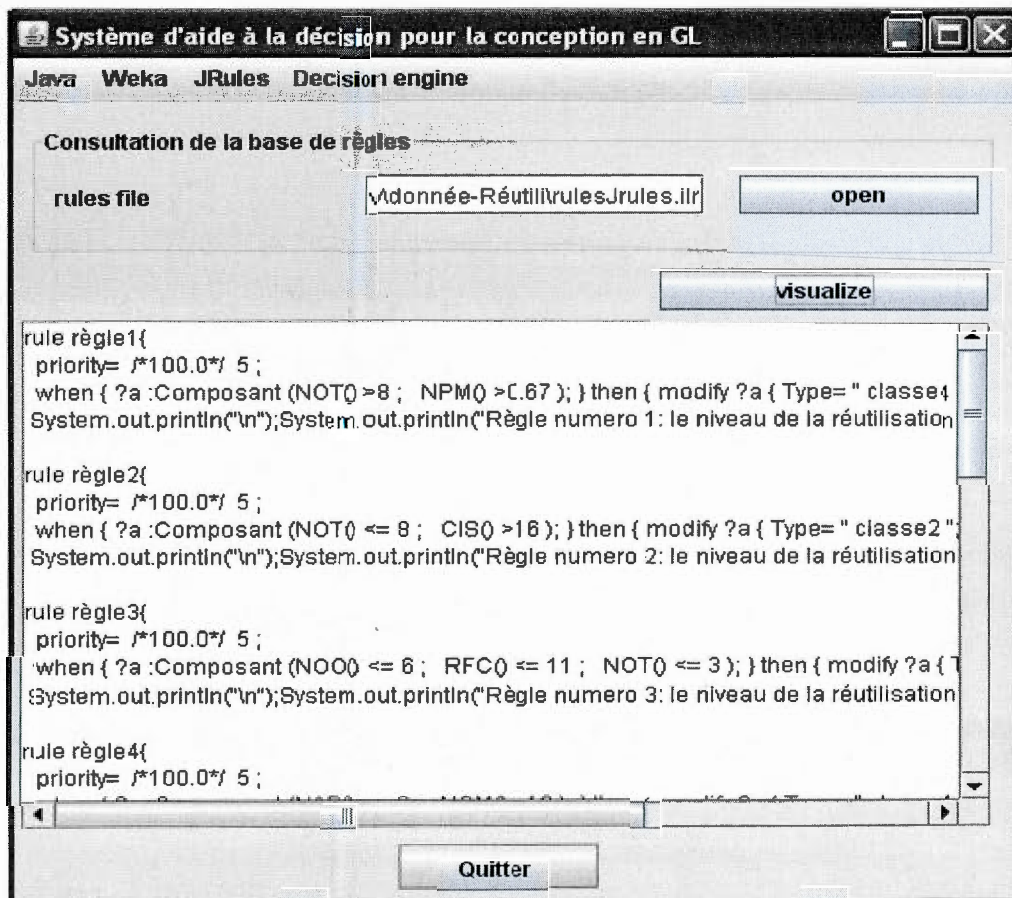

Figure 5.8 · Consultation d'une base de connaissance par un débutant

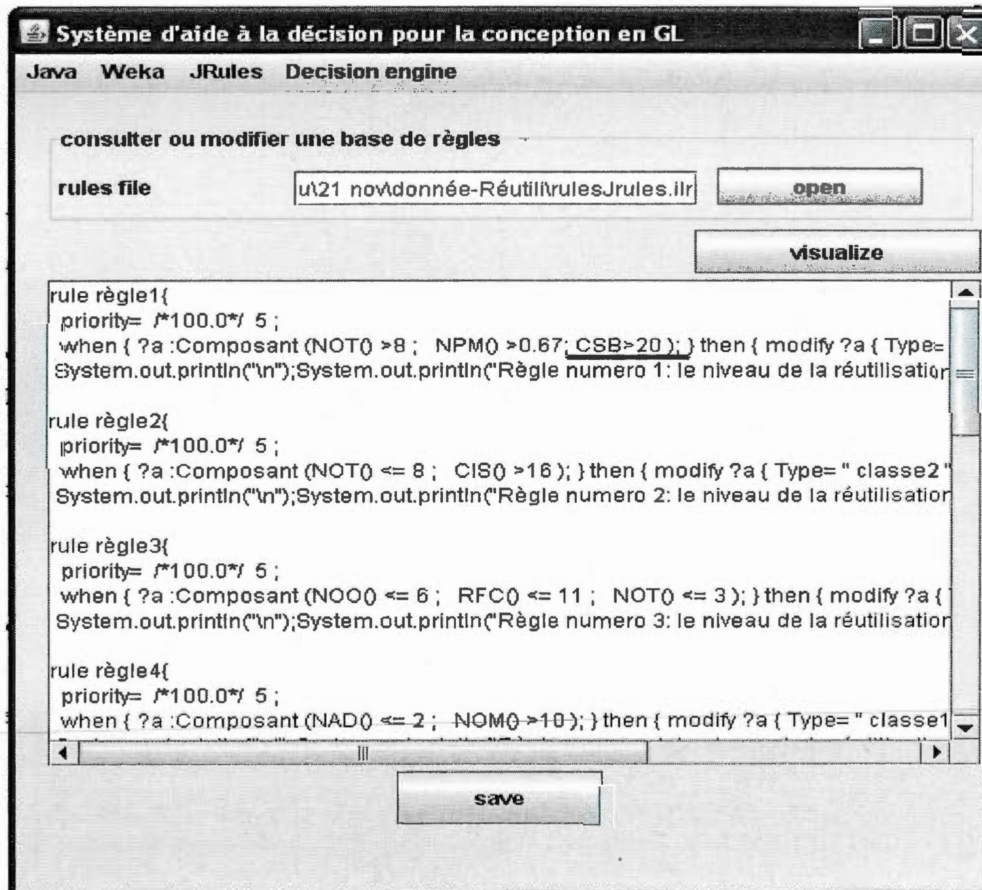

Figure 5.9 Consultation et/ou modification d'une base de connaissances par un expert

Un expert peut ajouter ou supprimer une condition dans une règle, il peut aussi changer l'action d'une règle ou même sa priorité. Dans cet exemple, à l'aide de cette interface, l'expert peut ajouter, par exemple, une condition à la 1ère règle (figure 5.9).

Exemple d'un règle produite par l'algorithme Part de WEKA :

```
rule règle1{
priority= 5 ;
when { ?a :Composant (NOT() >8 ; NPM() >0.67 ); }
```

```
then {modify?a{Type="classe4"; System.out.println("\n");System.out.println("Règle numero 1: le niveau de la réutilisabilité de l'objet " + ?a.nameObjet() + " appartient à " + ?a.Type());} } };
```

Un expert voit que si on ajoute la condition 'CSB>20' à cette règle, elle devient plus pertinente, il peut aussi changer la priorité de règle, donc cette dernière passe de priorité 5 à priorité 1.

La même règle modifiée par un expert :

```
rule règle1{

priority= 1 ;

when { ?a :Composant (NOT() >8 ; NPM() >0.67; CSB>20 );}

then {modify?a{Type="classe4"; System.out.println("\n");System.out.println("Règle numero 1: le niveau de la réutilisabilité de l'objet " + ?a.nameObjet() + " appartient à " + ?a.Type());} } };
```

À la fin d'une telle consultation ou modification, l'expert peut sauvegarder la nouvelle règle.

5.5.6 La prédiction

La fonctionnalité principale de notre outil est la prédiction. Cette tâche peut être faite soit par un débutant ou par un expert. La prédiction d'un facteur de qualité d'un composant logiciel ou d'une classe d'un code source est le fait de lui attribuer une classification.

L'utilité d'un modèle de prédiction est multiple. Dans le cas de la maintenance préventive, il peut déterminer quelles classes doivent être réécrites et de quelle façon. Il peut être utilisé aussi pour déterminer quelle classe est susceptible d'engendrer des fautes. Dans le cadre de notre expérimentation, notre but est d'identifier les composants potentiellement réutilisables dans une application future. Nous allons alors exploiter les bases de connaissances que nous avons générées lors de l'apprentissage pour faire la prédiction.

Nous devons, d'abord, préparer le format des objets à prédire. Chaque objet (ex., une classe ou un composant) possède la même structure : un ensemble de paires attribut-valeur. Ces paires sont les différentes mesures de métriques associées à la taille et à la complexité.

Contrairement à l'apprentissage, nous n'avons pas la valeur de l'attribut qui représente le niveau de réutilisabilité de l'objet. C'est en fait, cette valeur que nous cherchons à identifier lors de la prédiction.

Voici un exemple d'un ensemble d'objets de type « classe d'un code source » pour lesquels nous souhaitons prédire le niveau de réutilisabilité, à partir de leurs attributs internes de qualité relatifs à la taille et la complexité.

Class Name	NOM	RFC	CIS	NOT	NOP	NOD	NAD	NRA	NPA	NOO	NPM	CSB
Agenda	12	54	12	1	2	2	2	1	0	2	0,83	80
ArgsSet	15	26	15	2	0	2	1	1	0	4	0,87	8
BasicAgent	31	92	16	17	12	13	3	6	0	1	0,71	268
Belief	19	106	19	18	0	4	4	0	0	3	1,26	168
BeliefAction	14	24	14	6	5	3	3	0	1	4	0,71	96
BeliefCond	14	22	14	8	5	4	4	0	1	6	0,71	264
BeliefMonitor	17	27	17	6	1	3	3	1	0	4	2,18	116
BeliefsKB	11	29	11	0	0	2	1	0	0	1	1,45	80
BinaryTimeEx	20	51	20	4	10	3	2	2	1	2	0,75	8
CapabilitiesKB	15	64	15	2	0	3	2	0	0	1	2,27	156
Capability	12	29	12	4	0	3	2	0	0	2	0,67	204
CapabilityActi	12	11	12	5	5	2	2	0	0	3	0,58	56

Figure 5.10 Liste d'objets à prédire dans un fichier Excel

Dans un environnement normal de travail, les objets Java doivent être initialisés et stocker dans la mémoire de travail du moteur ILOG JRules; nous utilisons alors, l'action 'assert'. Si par exemple, nous voulons insérer dans la mémoire de travail une classe de nom 'C1', l'instruction est la suivante :

Assert classe (C1, m1, m2, m3, m4, m5, ...) ; où m1, m2, m3,... sont les mesures de métriques associées à cette classe par rapport à un attribut interne de qualité (couplage, héritage, complexité, taille).

Chaque règle dans ILOG JRules est écrite dans un fichier de règles. Ce fichier est d'extension '.ilr'. En voici un exemple :

```
import Package .*;
import java.util.* ;
setup
{
assert Classe ("C1", m1, m2, m3,..., mn) ;
assert Classe ("C2", m1, m2, m3,..., mn) ;
```

```
assert Classe ("C3", m1, m2, m3,..., mn) ;
....
} ;
rule règle1{........};
rule règle2{........};
rule règle3{........};
....
```

Les objets à prédire nous sont fournis dans un fichier Excel. Nous devons convertir le contenu de ce fichier en des instructions JRules spécifiques au fichier '.ilr' afin de les insérer automatiquement en phase de prédiction dans la mémoire de travail de JRules.

La figure ci-dessous montre les résultats de conversion des données du fichier Excel vers JRules.

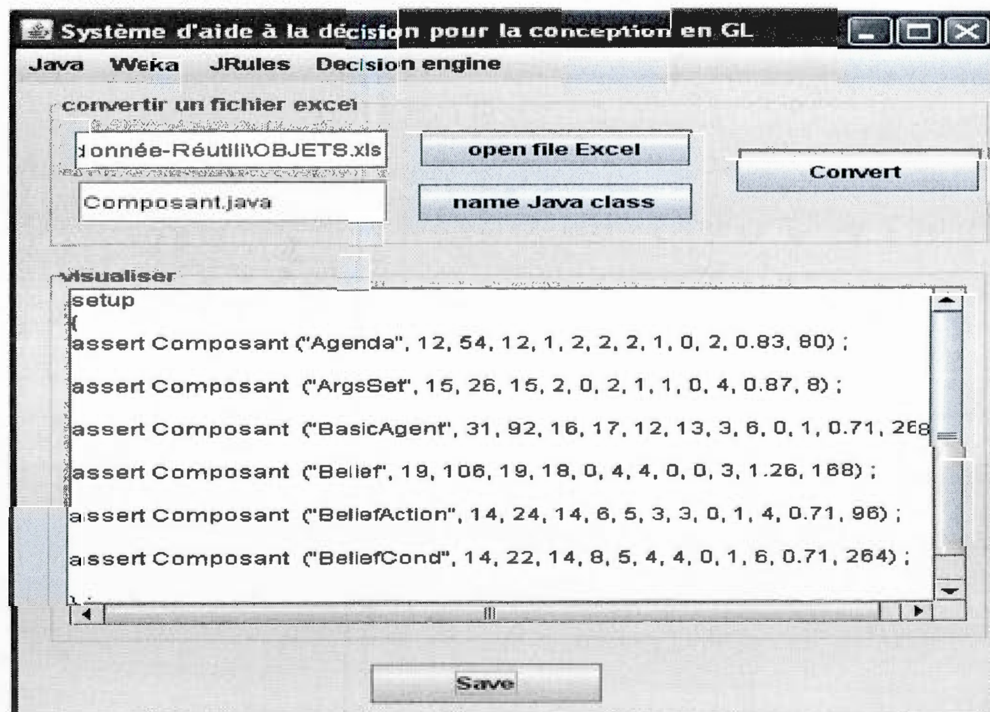

Figure 5.11 Exemple d'un ensemble d'objets à insérer dans la mémoire de travail de JRules

5.5.6.1 Description de l'interface de prédiction

L'exemple d'exécution dans la figure ci-dessous consiste à prédire le niveau de réutilisabilité de quatre composants logiciels. Ces objets sont décrits par leurs noms et les mesures de métriques associées à la taille et à la complexité.

La première partie du fichier '.ilr' qui commence par l'instruction 'Setup' permet d'insérer ces objets dans la mémoire de travail d'ILOG JRules. Nous avons déjà préparé ce code lors de la préparation des données (voir la section précédente). La deuxième partie est le fichier de règles, c'est le modèle de prédiction produit lors de l'apprentissage puis converti dans la syntaxe de JRules. Nous joignons à l'aide du bouton 'rule file' ces deux fichiers, pour obtenir la structure globale du modèle prêt à être exécuté.

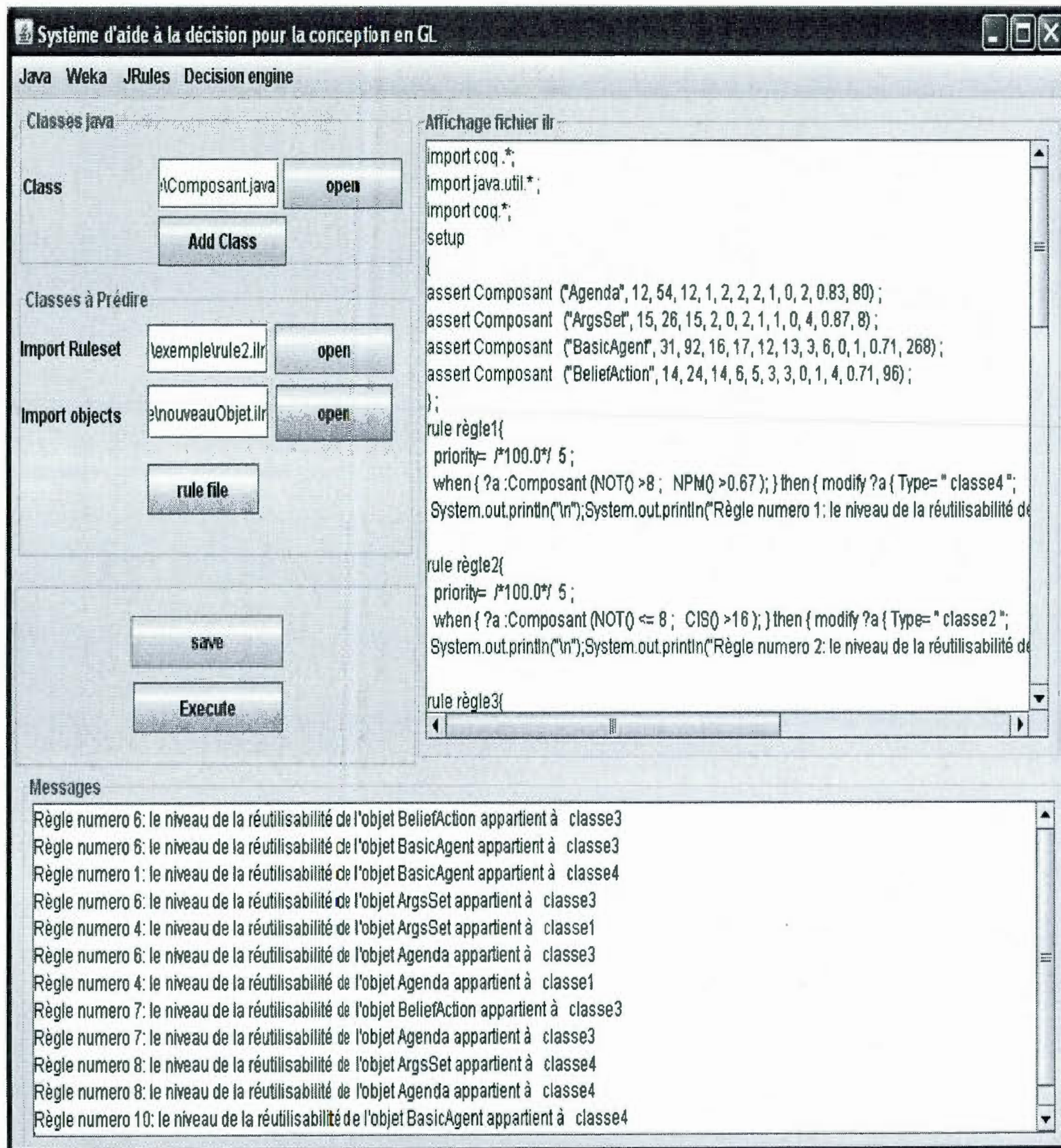

Figure 5.12 Exemple d'exécution d'un modèle pour prédire la réutilisabilité de composants logiciels

5.5.6.2 Analyse des résultats de prédiction

Le 'rule set' contient 4 objets à prédire et 11 règles à exécuter. Les règles sont:

```
rule règle1{
priority= /*100.0*/ 5 ;
when { ?a :Composant (NOT() >8 ;  NPM() >0.67 ); } then { modify ?a { Type= " classe4 ";
} };

rule règle2{
priority= /*100.0*/ 5 ;
when { ?a :Composant (NOT() <= 8 ;  CIS() >16 ); } then { modify ?a { Type= " classe2 ";}
};

rule règle3{
priority= /*100.0*/ 5 ;
when { ?a :Composant (NOO() <= 6 ;  RFC() <= 11 ;  NOT() <= 3 ); } then { modify ?a {
Type= " classe2 ";} };

rule règle4{
priority= /*100.0*/ 5 ;
when { ?a :Composant (NAD() <= 2 ;  NOM() >10 ); } then { modify ?a { Type= " classe1
";} };

rule règle5{
priority= /*100.0*/ 5 ;
when { ?a :Composant (NPM() <= 0.7 ); } then { modify ?a { Type= " classe2 "; } };

rule règle6{
priority= /*100.0*/ 5 ;
```

```
when { ?a :Composant ( ); } then { modify ?a { Type= " classe3 ";} };

rule règle7{
priority= /*94.117645*/ 4 ;
when { ?a :Composant (NOT() <= 8 ; CSB() >16 ; NOP() >0 ; NAD() <= 3 ; NAD() >0 );
} then { modify ?a { Type= " classe3 ";} };

rule règle8{
priority= /*93.10345*/ 3 ;
when { ?a :Composant (NPM() >0.77 ; NAD() <= 3 ; CSB() <= 324 ; CSB() >4 ); } then {
modify ?a { Type= " classe4 "; } };

rule règle9{
priority= /*90.0*/ 2 ;
when { ?a :Composant (NOP() <= 5 ; CIS() <= 12 ; NRA() <= 0 ); } then { modify ?a {
Type= " classe4 "; } };

rule règle10{
priority= /*80.0*/ 1 ;
when { ?a :Composant (NRA() >1 ); } then { modify ?a { Type= " classe4 "; } };

rule règle11{
priority= /*75.0*/ 0 ;
when { ?a :Composant (NOP() >5 ; NPA() <= 1 ; NPA() <= 0 ; NOT() <= 4 ; NOP() <= 7
); } then { modify ?a { Type= " classe1 "; } };
```

Les règles seront exécutées selon leurs priorités. Ils sont identifiés par des numéros. La règle numéro 6 est la règle par défaut. Les règles numéro 1, 2, 3, 4 et 5 possèdent la plus grande priorité et la règle numéro 11 a la plus faible priorité.

Le résultat d'exécution est la suivante :

Règle numero 6: le niveau de la réutilisabilité de l'objet BeliefAction appartient à classe3
Règle numero 6: le niveau de la réutilisabilité de l'objet BasicAgent appartient à classe3
Règle numero 1: le niveau de la réutilisabilité de l'objet BasicAgent appartient à classe4
Règle numero 6: le niveau de la réutilisabilité de l'objet ArgsSet appartient à classe3
Règle numero 4: le niveau de la réutilisabilité de l'objet ArgsSet appartient à classe1
Règle numero 6: le niveau de la réutilisabilité de l'objet Agenda appartient à classe3
Règle numero 4: le niveau de la réutilisabilité de l'objet Agenda appartient à classe1
Règle numero 7: le niveau de la réutilisabilité de l'objet BeliefAction appartient à classe3
Règle numero 7: le niveau de la réutilisabilité de l'objet Agenda appartient à classe3
Règle numero 8: le niveau de la réutilisabilité de l'objet ArgsSet appartient à classe4
Règle numero 8: le niveau de la réutilisabilité de l'objet Agenda appartient à classe4
Règle numero 10: le niveau de la réutilisabilité de l'objet BasicAgent appartient à classe4

On résume le résultat de la prédiction dans le tableau suivant :

classe / *Objet*	*Classe1*	*Classe2*	*Classe3*	*Classe4*
Agenda	R4		R6 /R7	R8
ArgsSet	R4		R6	R8
BasicAgent				R1 /R10
BeliefAction			R6 / R7	

Tableau 5.1 Résultats de classification des composants logiciels

- Le niveau de réutilisabilité du 1er objet ***'Agenda'*** appartient à la classe1 selon la règle 4, et appartient à la classe 3 selon les deux règles 6 et 7. Il appartient à la classe4 selon la règle 8. Dans ce cas, comme la règle 4 est la plus prioritaire, la classe à laquelle appartient cet objet serait la '*classe1*'. Cet objet est totalement réutilisable.

- Le niveau de réutilisabilité du 2ème objet ***'ArgsSet'*** appartient à la classe1 selon la règle 4, et appartient à la classe 3 selon la règle 6. Il appartient à la classe 4 selon la règle 8. Dans ce cas, comme la règle 4 est la plus prioritaire, la classe à laquelle appartient cet objet serait la '*classe1*'. Cet objet est totalement réutilisable.

- Le niveau de réutilisabilité du 3ème objet ***'BasicAgent'*** appartient à la classe4 selon les deux règles 1 et 10. Donc cet objet est de la '*classe4*'.Cet objet est non réutilisable.

- Le niveau de réutilisabilité du 4ème objet ***'BeliefAction'*** appartient à la classe3 selon les deux règles 6 et 7. La classe à laquelle appartient cet objet est la '*classe3*'. Cet objet est réutilisable avec de grands changements.

5.5.6.3 Validation d'un modèle de prédiction

C'est dans cette activité que les connaissances et les modèles de prédiction sont validées et vérifiées. Il est très important d'exécuter différents tests afin de valider les éléments de la base de connaissance. Pour ce faire, les résultats de tests et d'exécutions faites précédemment seront présentés aux experts pour en vérifier la précision. L'expert est le mieux placé pour valider ces résultats. À vrai dire, un expert pourrait éliminer, par exemple, la règle par défaut numéro 6, car elle est trop générale et inutile. Il pourrait aussi examiner les autres règles en les modifiant, soit en ajoutant un ou plusieurs tests à la condition de la règle, soit aussi en supprimant un. Tout ça dépend de son niveau d'expertise et de ses connaissances. L'expert est le seul utilisateur qui a le droit de faire ces améliorations et ces mises à jour aux modèles de prédiction. Il utilise pour cela le «le bon sens » et l'intuition afin

de prendre une telle décision et valider un modèle de prédiction. Le but est d'obtenir des modèles plus performants et plus précis pour l'estimation de la qualité.

5.6 Conclusion

La prédiction de la qualité constitue la fonctionnalité principale de notre outil, nous lui avons accordée une place importante. Les résultats de prédiction seront plus utiles si la quantité de données utilisée est plus importante. Ainsi l'absence de connaissances proposées par des experts du développement logiciel réduit la performance des modèles de prédiction. De plus, le moteur d'inférence de JRules fonctionne selon une stratégie bien définie; il serait intéressant d'explorer d'autres stratégies de déclenchement des règles.

CONCLUSION

L'objectif ultime de notre recherche est la prédiction de la qualité d'un nouveau produit logiciel. Pour cela nous avons développé un prototype d'un système d'aide à la décision qui nous permet d'estimer et de prédire des facteurs de qualité de composants logiciels afin d'améliorer le processus de développement. C'est en se basant sur des mesures faites sur des composants logiciels fonctionnels, que nous avons tiré les connaissances nécessaires et utiles pour réaliser nos expérimentations.

Notre outil nous permet de vérifier certaines hypothèses portant sur l'existence de relations entre les facteurs de qualité dont la réutilisabilité, la propension à engendrer des erreurs et la maintenabilité d'une part, et les attributs internes que sont : l'héritage, la cohésion, le couplage, la taille et la complexité, d'autre part.

Nous avons abordé la problématique selon trois thèmes :

- la qualité d'un produit logiciel qui représente le cadre général de notre mémoire,
- les techniques d'apprentissage automatique et la capitalisation des connaissances,
- l'estimation et la prédiction de la qualité.

Nous avons commencé à décrire la qualité du produit logiciel par la définition des concepts de base tels que les facteurs de qualité, les attributs internes et externes ainsi que les métriques liées à chaque attribut interne de qualité. En deuxième lieu, nous avons effectué une étude sur les méthodes de gestion et d'acquisition des connaissances en génie logiciel. Nous avons abordé l'apprentissage automatique et en particulier l'apprentissage des règles qui représente une approche efficace permettant de créer de nouveaux modèles prédictifs et de découvrir de nouvelles connaissances. Enfin, l'accent a été mis sur le concept de

prédiction et de prise de décision qui est la base de ce mémoire. À l'aide d'un moteur d'inférence et de règles, les utilisateurs peuvent détecter les problèmes liés aux composants logiciels, prédire la qualité et prendre des décisions sur la conception et sur le choix de développement.

Nous avons, alors, développé une plateforme d'un système d'aide à la décision regroupant les fonctionnalités de Weka et JRules. Nous avons implémenté les différentes interfaces à l'aide d'Éclipse. Plusieurs tâches peuvent être réalisées à l'aide de notre prototype telles que : convertir un fichier 'Excel' en un fichier 'arff', l'apprentissage pour deux types d'utilisateurs (expert et débutant), la création, la consultation et la mise à jour d'une base de connaissances (règles) et la prédiction en exécutant sur de nouvelles données les modèles générés par l'apprentissage.

L'élément central du système d'aide à la décision est la base de connaissances. La qualité des connaissances influe grandement sur la façon de raisonner et d'estimer une problématique. Pour cela, nous avons défini deux types de connaissances : les connaissances humaines (experts) d'une part et les connaissances provenant de l'apprentissage automatique. Ensuite, nous avons décrit comment une base de connaissances peut être jumelée à un moteur d'inférence qui permet de raisonner et de donner des résultats pertinents sut la prédiction et l'estimation. Dans notre prototype, nous avons utilisé le moteur d'inférence de JRules. Ce moteur parcourt la base de connaissances en décidant quelles règles correspondent bien au problème.

Cependant, la prédiction reste une problématique empirique en génie logiciel. Nous avons besoin de données pour représenter, créer, analyser et valider les modèles. Les données sont cruciales pour construire d'abord, puis valider les modèles de prédiction. Mais, malheureusement, nous ne pouvons atteindre un haut niveau de maturité de la prédiction à cause du manque de données. Nous ne trouvons pas assez de données en raison de la confidentialité que de nombreuses organisations imposent; elles interdisent l'accès à leurs bases de données, par crainte de perdre leurs propres bases de connaissances.

Afin d'améliorer ce travail, nous proposons notamment d'ajouter les fonctionnalités suivantes à notre prototype:

- utiliser des techniques d'apprentissage autre que l'apprentissage de règles afin de générer plus de modèles prédictifs.
- faire la prédiction en utilisant plusieurs modèles en même temps.
- comparer les différents résultats de prédiction.

BIBLIOGRAPHIE

Abdel-Ghaly A.A., Chan P.Y. et Littlewood B., (1986). *Evaluation of competing software reliability predictions.* IEEE Transactions on Software Engineering, vol 12, p.950–967.

Abdi M.K., Lounis H. et Sahraoui H., (2006). *Analyzing Change Impact in Object-Oriented Systems.*

Abdi M. K., Lounis H. et Sahraoui H., (2009). *Predicting Change Impact in Object-Oriented Applications with Bayesian Networks*

Abreu, Fernando B. et Rogério C., (1994). *Candidate Metrics for Object- Oriented Software within a Taxonomy Framework.* Journal of Systems and Software, North-Holland, Elsevier Science, vol 26, p. 87-96.

Almeida M., Lounis H. et Melo W. L., (1999). *An Investigation on the Use of Machine Learned Models for Estimating Software correctability.* International Journal of Software Engineering and Knowledge Engineering.

Almeida M.A., Lounis H. et Melo W.L., (1998). *An Investigation on the Use of Machine Learned Models for Estimating Correction Costs.*

Bailly C., Challine J.F., Gloess P.Y., Ferri H.C. et Marchesin B., (1987). *Les langages orientés objet . concepts, Langages et Applications,* Cepadues édition.

Barbara K., (1996) *Software quality: the elusive Target,* national Computing Centre, Shari Lawrence Pfleeger, Systems/Softaware, Inc, vol 13, p. 12-21.

Bell D., Morrey I. et Pugh J., (1992). *Software Engineering: A Programming Approach,* Prentice Hall Publisher.

Boehm B., (1978). *Characteristics of Software Quality.* TRWSeries of Software Technology.

Boukadoum M., Sahraoui H. et Lounis H., (2001). *Machine Learning Approach to Predict Software Evolvability using Fuzzy Binary Trees.* In Proc. of Internatinal Conference on Artificial Intelligence.

Breiman L., Friedman J., Olshen R. et Stone C., (1984). *Classification and Regression Tree,* California: Wadsworth International.

Briand L.C., Wüst J., Ikonomovski S. et Lounis H., (1999). *Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study.*

Briand, Lionel, John D. et Jurgen W., (1997). *A unified framework for cohesion measurement in object-oriented systems.* In Proceedings of 4th International Software Metrics Symposium (ISMS). p. 43-53.

Briand, Lionel, Prem D. et Walcelio M., (1997). *An Investigation into Coupling Measures for C++.* In Proceedings of 19th International Conference on Software Engineering (ICSE). p. 412-421. Boston, USA.

Chidamber S., Kemerer C., (1994). *A metric suite for object oriented design.* IEEE Transactions on software engineering, vol 20, p. 476-493.

Chidamber S. et Kemerer C., (1991). *Towards a Metrics Suite for Object-Oriented Design.* In Conference Proceedings on Object-Oriented Programming, Systems, Languages and Applications. p. 197- 211.

Clark, Peter et Robin B., (1991). *Rule Induction with CN2: Some Recent Improvement.* p. 151–163.

Clark, Peter et Tim N, (1988). *The CN2 Induction Algorithm. Machine Learning Journal,* p. 261–283.

Ermine, J.L., (2001). *Ingénierie et capitalisation des connaissances*. Paris : Hernèse : science publication : chapitre 4, p. 65-102.

Ermine, J.L., (2003). *La gestion des connaissances*. Paris : Herme science : Lavoisier, p. 166.

Fenton N. et Pfleeger S., (1997). *Software Metrics A rigorous & Practical Approach*. London: Cmabridge Unversity Press, Cambridge.

Frank E. et Witten I.H., (1998). *Generating Accurate Rule Sets Without Global Optimization*. In Shavlik, J., ed., Machine Learning: Proceedings of the Fifteenth International Conference, Morgan Kaufmann Publishers, San Francisco, CA.

Fürnkranz J., (1999). *Separate-and-Conquer Rule Learning*. Artificial Intelligence Review. p.3-54.

Gillies A.C., (1992) *software quality, theory and management*, chapman & hall, p. 19-40.

ISO/IEC 9126-1 (2001) Software engineering --Product quality --*Part 1: Quality model*. International Organization for Standardization, International Electrotec1mical Commission.

ISO/IEC TR 9126-2 (2003) Software engineering --Product quality --*Part 2: External metrics*. International Organization for Standardization, International Electrotechnical Commission.

ISO/IEC TR 9126-3 (2003) Software engineering --Product quality --*Part 3: Internal metrics*. International Organization for Standardization, International Electrotechnical Commission.

ISO/IEC TR 9126-4 (2004) Software engineering --Product quality --*Part 4: Quality in use metrics*. International Organization for Standardization, International Electrotechnical Commission.

Jacquet J.P. et Abran A., (1997), *From Software Metrics to Software Measurement Methods: A Process Model*. Third International Symposium and Forum on Software Engineering Standards, Walnut Creek.

Lee Y.S., Liang B.S., Wu S.F. et Wang F.J., (1995). *Measuring the Coupling and Cohesion of an Object Oriented Program Based on Information Flow*. In Proceedings of Internationnal conference on Software Quality,Maribor, Slovenia.

Li W. et Henry S., (1993). *Maintenance metrics for the object oriented paradigm*. In Proceedings of the First International Software Metrics Symposium. p. 52-60. Baltimore, Maryland.

Li W. et Henry S., (1993). *Object oriented metrics that predicts maintainability*. System and Software,

Lorenz M. et Kidd J., (1994). *Object-oriented software metrics: a practical guide*. États-Unis: Englewood Cliffs, N.J.: PTR Prentice Hall, A Pearson Education Company. p. 146.

Lounis H., Abdi M.K. et Sahraoui H., (2009). *Predicting Maintainability expressed as Change Impact with Machine-Learning techniques.*

Lounis H., Abdi M.K. et Sahraoui H., (2009). *Inducing Knowledge for Change Impact Analysis.*

Lounis H., Ait-Mahiedine L., (2004). *Machine Learning-Based Quality Predictive Models: Towards an Artificial Intelligence Decision Making System.*

Lounis H., Sahraoui H. et walcelio L., (1998). *Vers un modèle de prédiction de la qualité du logiciel pour les systèmes à objets.*

Manuel Z. et Mickel G., (2001) *Management des connaissances, modèles d'entreprise et application sous la direction*, p. 16.

Mao Y., (1998), *A metric based detection of reusable object-oriented software components using machine learning algorithm*, Master thesis, Mc Gill University, Montreal.

McCall J.A., Richards P.K. et Walters G.F., (1977). *Factors in software quality.* Technical report, US Rome Air Development Center Reports.

Michalski R.S., (1969). *On the Quasi-Minimal Solution of the General Covering Problem,* Proceedings of the V International Symposium on Informatoon Processing (FCIP 69), Vol. A3 (Switching Circuits), Bled, Yugoslavia, p.125-128

Milicic D., (2005) *Software quality attributes and trade-offs*, chapter 1 -Software Quality Models and Philosophies.

Mitchell T.M., Carbonell J.G. et Michalski R.S..(1986). *Machine learnin : A Guide to current Research.* Kluwer Academic Publishers .p.194.

Mitchell T.M., (1997). *Machine Learning*, McGraw-Hill

Nonaka I., (1994). *Dynamic theory of organizational Knowledge Creation.* Organizational science, vol.5, p. 14-37

Paul E.U., (1989). *Incremental Induction of Decision Trees.* Journal Machine Learning, vol4 , p. 161-186

Paul M.E., (2001). *Knowledge Management in software Engineering, A State of the Art Report.*

Paul O. et Pfleeger S.L., (1997). *Applying Software Metrics*, IEEE Computer

Philippe D., (2005). *Mesure de complexité et maintenance de code.*

Porter A. et Selby R., (1990). *Empirically-Guided Software Development using Metric-Based Classification Trees.* IEEE Software, vol 7, p. 46-54.

Pressman et Roger S., (1997). *Software engineering: a practitioner's approach.* 4th ed. Édition New York, N.Y.: McGraw-Hill.

Pressman et Roger S., (2004). *Software Engineering: A Practitioner's Approach,* 6th ed. New York, NY, USA: McGraw-Hill, p. 880.

Quinlan J.R. et Cameron-Jones R.M., (1993). *FOIL: A midterm report.* In proceedings of the European Conference on Machine Learning, p. 3-20.

Quinlan J.R, (1993), *C4.5: Programs for Machine Learning,* Morgan Kaufmann (Ed.). p. 302.

Scott A., (1998). *A Realistic Look at Object-Oriented Reuse. Journal Software Development, vol 6, p. 30-38,*

William W.C., (1995). F*ast effective rule induction.* In: twelfth international conference on machine learning, vol 3, p. 115-123.

Yazid H. et Lounis H., (2006). *Exploring an Open Source Data Mining Environment for Software Product Quality Decision Making.*

Zacklad M. et Grundstein M.., (2001). *Management des connaissances.* Paris : Hermès, p. 245.