UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À L'UNIVERSITÉ DU
QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
LAZHAR SADAOUI

ÉVALUATION DE LA COHÉSION DES CLASSES : UNE NOUVELLE APPROCHE BASÉE SUR LA CLASSIFICATION

Juin 2010

ÉVALUATION DE LA COHÉSION DES CLASSES : UNE NOUVELLE APPROCHE BASÉE SUR LA CLASSIFICATION

Lazhar Sadaoui

SOMMAIRE

La cohésion est considérée comme étant l'un des attributs les plus importants des systèmes orientés-objet. Plusieurs métriques ont été proposées dans la littérature pour évaluer la cohésion des classes. Une classe fortement cohésive est plus facile à comprendre, à maintenir et à réutiliser. Dans le paradigme objet, une classe est cohésive quand ses différentes parties sont fortement reliées (contribuent à son implémentation). Il devrait être difficile de diviser une classe cohésive. Une classe à faible cohésion possède des membres disparates et non reliés. La cohésion peut être utilisée pour identifier les classes mal conçues.

La majorité des métriques existantes, dites structurelles, essaient de capturer la cohésion d'une classe en termes de connexions parmi ses membres. Elles comptent le nombre de variables d'instance utilisées par les méthodes ou le nombre de paires de méthodes partageant des variables d'instance. Ces métriques échouent dans plusieurs situations à refléter proprement la cohésion des classes, particulièrement en termes de méthodes reliées. Elles ne tiennent pas compte de certaines caractéristiques des classes telles que, par exemple, la taille des parties cohésives et la connectivité parmi les membres d'une classe. Par ailleurs, les métriques structurelles donnent peu d'indications sur les aspects conceptuels d'une classe. Elles ne sont pas tout à fait capables de détecter efficacement des anomalies de conception liées, par exemple, à l'affectation de responsabilités disparates.

La classification est une technique permettant de mieux distinguer les objets et les classer selon leur similarité. Son but est de regrouper les éléments similaires ou reliés ensemble et de séparer les groupes disjoints. Nous pensons alors qu'il est possible de l'utiliser pour séparer les groupes cohésifs pouvant exister à l'intérieur d'une classe et améliorer l'évaluation de sa cohésion. Ce travail propose donc une nouvelle approche de calcul de la cohésion des classes basée sur la technique de classification. Le principe de cette technique repose sur les similarités pouvant exister entre les membres d'une classe. Grâce au potentiel de la classification, il est possible de déterminer les regroupements cohésifs de méthodes au sein d'une classe. Cela va certainement aider à déceler la disparité dans le code et les problèmes liés à l'affectation de responsabilités disparates aux classes d'une manière générale.

L'approche proposée dans ce travail repose spécifiquement sur la classification hiérarchique avec le coefficient de Jaccard comme mesure de similarité et a été évaluée

sur différents exemples concrets de classes présentant des problèmes de conception liés soit à la disparité du code et/ou à la présence de code correspondant à des préoccupations transverses. Les valeurs de cohésion obtenues grâce à la nouvelle approche, comparées aux valeurs données par des métriques structurelles choisies dans la littérature, reflètent mieux la cohésion (et la structure) des classes évaluées. Elles fournissent une meilleure indication sur la présence de disparité dans le code. La nouvelle approche pourrait, contrairement aux approches structurelles existantes, être utilisée pour supporter ou orienter des actions de restructuration des classes (aspect mining et refactoring, entre autres).

CLASS COHESION MEASUREMENT: A NEW APPROACH BASED ON CLUSTERING

Lazhar Sadaoui

ABSTRACT

Cohesion is considered as one of the most important attributes of object-oriented systems. Many metrics have been proposed in the literature for class cohesion measurement. A cohesive class is easier to understand, to maintain and to reuse. In the objet paradigm, a class is cohesive when its parts are highly related (connected), all its parts should contribute to its realization. It should be difficult to split a cohesive class. A low cohesive class has disparate members. Cohesion can be used to identify poorly designed classes.

The majority of existing metrics (called structural metrics) try to capture the cohesion of a class in terms of connections between its members. They count the number of instance variables used by methods or the number of pairs of methods sharing instance variables. These metrics fail in several situations to properly reflect the cohesion of classes, particularly in terms of connected methods. They do not take into account some characteristics of classes. Moreover, structural metrics give few indications on the conceptual aspects of a class. So they are not able to efficiently detect design problems related, for example, to the assignment of disparate responsibilities to classes.

Clustering is a technique allowing a better differentiation and classification of objects according to their similarity. Then, we believe that using clustering techniques will allow the separation of cohesive groups that can exist into a class and hence to improve the measurement of class cohesion. This work proposes a new approach to evaluate the cohesion of classes based on clustering. The principle of this technique is based on the similarities that can exist between class members. Thanks to the potential of clustering, it's possible to determine the cohesive groups of methods within a class. This will certainly assist in revealing disparity in the code and problems related to the disparate assignment of responsibilities to the classes of a system.

The approach proposed in this work is specifically based on the hierarchical clustering with the Jaccard coefficient as a measure of similarity and has been evaluated on various concrete examples of classes presenting some design problems such as the disparity of the code and/or the presence of code related to crosscutting concerns. Cohesion values given by the new approach, compared with values given by structural metrics chosen from literature, reflect better the cohesion (and the structure) of assessed classes. They provide a better indication on the presence of disparity in the code. The

new approach could be, contrary to structural approaches, used to support (or guide) some activities of restructuring of classes (aspect mining and refactoring, among others).

REMERCIEMENTS

Je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à la réalisation de ce travail. Tout d'abord, je souhaiterais manifester ma reconnaissance et ma gratitude particulièrement à mes directeurs de recherche les professeurs Linda BADRI et Mourad BADRI de m'avoir accueilli dans leur équipe de recherche, et d'avoir accepté de diriger mon travail. Je ne saurais suffisamment les remercier pour l'inspiration, l'aide et le temps qu'ils ont bien voulu me consacrer dans les moments difficiles malgré, la multiplicité de leurs activités. Sans leurs conseils judicieux, ce mémoire n'aurait jamais vu le jour. Je les remercie aussi d'avoir su me laisser la liberté de traiter le sujet et me permettre de mieux délimiter le cadre d'un sujet qui aurait pu s'avérer inabordable dans le temps imparti. Sans oublier pour autant la motivation et le courage qu'ils m'ont donné pour me lancer dans un projet de Maîtrise après une longue interruption d'étude!

Je tiens à remercier chaleureusement tous les professeurs que j'ai eu l'occasion de côtoyer tout au long de mes études à l'UQTR au département de mathématiques et d'informatique, et qui ont su m'apporter les bases nécessaires à la réalisation de ce travail de recherche.

Enfin, merci à tous mes collègues du laboratoire de recherche en génie logiciel qui ont été mes compagnons de travail durant cette riche période de formation et aussi pour l'échange d'opinions.

DÉDICACES

À mes chers parents

À mon épouse

Et à mes deux enfants : Sheyma et Sami

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE I

INTRODUCTION

I.1. Contexte

Les travaux sur l'évolution des logiciels ont permis l'identification de plusieurs lois, en particulier, celles relatives au changement continuel, à la complexité croissante et à la dégradation de la qualité. Par ailleurs, le succès d'un projet ne se mesure pas uniquement à sa capacité à livrer une première version opérationnelle, mais également à sa capacité à évoluer à travers le temps pour supporter les nouveaux besoins. La maintenance est donc, plus que jamais une activité incontournable. Aussi, chaque fois qu'un logiciel est modifié, cela peut avoir un impact, qui peut être important, sur sa qualité. Les métriques constituent un moyen incontournable pour assister les tâches des différentes activités du Génie Logiciel, en particulier celles relatives à la maintenance, du fait qu'on ne peut pas contrôler ce qu'on ne peut pas mesurer [DeMarco 82]. La cohésion des classes est considérée comme étant l'un des attributs les plus importants des systèmes orientés-objet. Un composant fortement cohésif est plus facile à maintenir et à réutiliser [Bieman 95, Briand 98, Chae 00, Li 93]. C'est dans ce contexte que se situe le présent travail de recherche.

I.2. Problématique

Plusieurs métriques (dites structurelles) ont été proposées afin de mesurer la cohésion d'une classe dans les systèmes orientés-objet. Néanmoins, la cohésion a plusieurs aspects et reste beaucoup plus difficile à capturer que ce que suggèrent ces métriques. En effet, ces métriques ont plusieurs limites et manquent souvent pour certaines d'entre elles d'argumentation. Nous croyons que la cohésion des classes est un attribut logiciel complexe caractérisé par différents aspects pouvant être de nature structurelle, fonctionnelle, conceptuelle ou sémantique. En effet, deux méthodes d'une classe peuvent être reliées de façons différentes. Elles peuvent être reliées structurellement sans l'être sémantiquement par exemple. Les métriques existantes échouent dans plusieurs situations à refléter proprement ces propriétés et échouent généralement à évaluer la cohésion d'une classe avec précision. D'ailleurs, elles ne sont pas tout à fait capables de détecter efficacement des anomalies de conception liées, par exemple, à l'affectation de responsabilités disparates à des classes ou à la présence de code correspondant à des préoccupations transverses dans une classe. De ce fait, elles ne constituent par réellement un moyen efficace pour supporter des activités de maintenance, de restructuration ou d'aspect mining en tant que tel à moins d'être réajustées et adaptées.

I.3. Démarche

Dans le paradigme objet, mesurer la cohésion d'une classe consiste à mesurer le degré de liaison entre ses membres [Bieman 95]. Dans le contexte d'une implémentation orientée-objet, le mot liaison signifie souvent implicitement similarité des méthodes définies au niveau de la classe. Cela signifie que si les méthodes membres d'une classe fournissent une fonctionnalité similaire, alors la classe est dite cohésive et si elles fournissent des fonctionnalités non similaires alors la classe est dite moins cohésive ou non cohésive [Imran 04].

Le but de la classification est de différencier les groupes cohésifs au sein d'un ensemble d'objets donné, selon un ensemble de caractéristiques ou d'attributs appropriés des objets analysés. Chacun de ces sous-ensembles (clusters) représente les objets qui sont similaires entre eux-mêmes et dissemblables vis-à-vis des objets des autres groupes [Moldovan 06b]. La technique de classification a le potentiel de regrouper les objets similaires ensemble et de séparer et différencier les groupes distincts. À l'aide des techniques de classification, nous pensons être en mesure d'améliorer l'évaluation de la cohésion des classes, de sorte à mieux refléter la qualité conceptuelle de la classe, de déterminer les différents groupes « conceptuellement » cohésifs au sein d'une classe, de différencier les groupes de code disparates et ainsi déceler des problèmes liés à l'affectation de responsabilités disparates aux classes. Ceci nous permettra de déceler, éventuellement, la présence de code pouvant correspondre à des préoccupations transverses (réduisant la cohésion de la classe). Sauf qu'il reste à déterminer comment mettre en œuvre la technique pour le réaliser.

Notre travail vise donc à proposer une nouvelle approche permettant d'évaluer la cohésion des classes en se basant sur les techniques de classification. Dans cette étude, nous nous sommes intéressés spécifiquement à la classification hiérarchique avec comme mesure de similarité le coefficient de Jaccard, comme une première investigation.

Nous pensons que cette approche permettra de mieux identifier les classes faiblement conçues. Cette approche pourrait, par ailleurs, être concrètement utilisée pour supporter des activités d'aspect mining (détection de préoccupations transverses dans du code objet), orienter des actions de « refactoring aspect » (restructuration des classes en utilisant les aspects) et supporter ainsi l'évolution des systèmes orientés-objet vers des systèmes orientés-aspect.

I.4. Organisation du mémoire

Le reste de ce mémoire est organisé comme suit :

Au chapitre II, nous proposons une revue de la littérature sur la cohésion des classes, particulièrement dans les systèmes orientés-objet où nous avons souligné les principales limites des approches structurelles existantes et mis l'accent sur les critères de cohésion. Le chapitre III présente les concepts de base et le principe de la classification ainsi que ses différentes méthodes. Une seconde revue de la littérature relative aux travaux connexes relatifs à la classification dans les domaines de restructuration de programmes et de l'aspect mining est présentée dans le chapitre IV. C'est au chapitre V que nous décrivons notre nouvelle approche d'évaluation de la cohésion. Le chapitre VI décrit les différentes expérimentations réalisées pour évaluer notre proposition ainsi que les résultats que nous avons obtenus. Enfin, une synthèse de ce travail est présentée au niveau de la conclusion.

CHAPITRE II

LA COHÉSION DES CLASSES

II.1. Introduction

Le fondement théorique des approches pour la mesure de la cohésion se base sur l'ontologie des objets décrite par Bunge (1977-79) dans laquelle, la similarité des choses est définie comme l'ensemble des propriétés qu'ils ont en commun. Les premiers auteurs Chidamber et Kemerer [Chidamber 91] ont adapté cette idée pour définir la cohésion d'une classe en se basant sur le degré de similarité entre ses méthodes. Plusieurs métriques ont été proposées par la suite pour mesurer la cohésion des classes dans les systèmes orientés-objet. La plupart d'entre elles se sont inspirées de la métrique LCOM (Lack of Cohesion in Methods) définie par Chidamber et al. [Chidamber 91, Chidamber 94, Chidamber 98]. Elles capturent, chacune à sa manière, la cohésion en termes de connexions entre les membres d'une classe. La plupart (figure II.1) ont été présentées en détail et catégorisées dans [Briand 98]. Il y a eu plusieurs travaux d'évaluation, de redéfinition et d'amélioration de ces métriques.

La cohésion des classes est considérée comme étant l'un des attributs les plus importants des systèmes orientés-objet. Un composant fortement cohésif est plus facile à maintenir et à réutiliser [Bieman 95, Briand 98, Chae 00, Li 93]. Une forte cohésion demeure une propriété souhaitable pour un composant logiciel. La cohésion est le degré de liaison entre les membres d'un composant. Elle est forte lorsque le composant

implémente une seule fonction logique. Toutes ses parties devraient contribuer à cette implémentation.

Metric	Definition
LCOM1	Lack of cohesion in methods. The number of pairs of methods in the class using no instance variables in common.
LCOM2	Let P be the pairs of methods without shared instance variables, and Q be the pairs of methods with shared instance variables. Then LCOM2 = $\|P\| - \|Q\|$, if $\|P\| > \|Q\|$. If this difference is negative, LCOM2 is set to zero.
LCOM3	Consider an undirected graph G, where the vertices are the methods of a class, and there is an edge between two vertices if the corresponding methods share at least one instance variable. Then LCOM3 = | connected components of G |
LCOM4	Like LCOM3, where graph G additionally has an edge between vertices representing methods Mi and Mj, if Mi invokes Mj or vice versa.
Co	Connectivity. Let V be the vertices of graph G from LCOM4, and E its edges. Then $Co = 2 . \frac{\|E\| - (\|V\| - 1)}{(\|V\| - 1) . (\|V\| - 2)}$
LCOM5	Consider a set of methods {Mi} (I = 1, ... , m) accessing a set of instance variables {Aj} (j = 1, ..., a). Let μ (Aj) be the number of methods that reference Aj. Then $LCOM5 = \frac{(1/a) \Sigma_{1 \le j \le a} \mu (Aj) - m}{1 - m}$
Coh	A variation on LCOM5. $Coh = \frac{\Sigma_{1 \le j \le a} \mu (Aj)}{m . a}$
TCC	Tight Class Cohesion. Consider a class with N public methods. Let NP be the maximum number of public method pairs : NP = [N * (N – 1)] / 2. Let NDC be the number of direct connections between public methods. Then TCC is defined as the relative number of directly connected public methods. Then, TCC = NDC / NP.
LCC	Loose Class Cohesion. Let NIC be the number of direct or indirect connections between public methods. Then LCC is defined as the relative number of directly or indirectly connected public methods. LCC = NIC / NP.

Figure II.1 : Principales métriques de cohésion existantes [Briand 98].

Dans le paradigme objet, une classe est cohésive lorsque ses parties sont hautement reliées. Il devrait être difficile de diviser une classe cohésive. Cependant, une modélisation malpropre lors de la conception peut produire des classes à faible cohésion. Ces dernières auront des membres disparates et non reliés. La cohésion peut être utilisée pour identifier les classes mal conçues. Un composant présentant une faible cohésion peut être formé de membres disparates et non reliés. Plusieurs responsabilités non reliées sont généralement affectées à de tels composants. Les éléments de conception présentant une faible cohésion devraient être considérés pour une éventuelle restructuration [Gelinas 05].

II.2. Les critères de cohésion

La plupart des métriques capturent la cohésion en termes de connexions entre les membres d'une classe. Le principe derrière cette catégorie de métriques est de mesurer le couplage (ou connectivité) entre les méthodes d'une classe. En fait, deux méthodes peuvent être connectées de différentes manières selon les critères suivants, que nous détaillerons par la suite:

1. Usage d'attributs
2. Invocation de méthodes
3. Arguments de type objet communs

Les métriques de cohésion se basent soient sur le premier critère (pour une grande majorité), soit sur les deux premiers. D'autres considèrent les trois critères (parfois à leur façon). Nous donnons, dans ce qui suit, une définition des trois critères tels que définis dans l'approche développée par Badri et al. [Badri 04, Badri 08].

Certains des éléments sur lesquels se basent ces derniers travaux ont été proposés par d'autres auteurs. Le but ici étant surtout de donner une idée sur ce type de critères pour illustrer les approches dites structurelles (sur quoi elles se basent), et pouvoir par la suite tenir une comparaison avec la nouvelle proposition de ce mémoire.

II.2.1. Critère d'usage d'attributs (UA)

Définition 1[Badri 08] **:** Considérons une classe C. Soit A= $\{A_1, A_2 \ldots. A_a\}$ l'ensemble des ses attributs et M = $\{M_1, M_2 \ldots, M_n\}$ l'ensemble de ses méthodes. Soit UA_{Mi} l'ensemble de tous les attributs utilisés directement ou indirectement par une méthode M_i. Un attribut est utilisé directement par une méthode M_i, si l'attribut apparaît dans le corps de la méthode M_i. Un attribut est utilisé indirectement par une méthode M_i s'il est utilisé directement par une autre méthode de la classe qui est invoquée directement ou indirectement par M_i. Il y a n ensembles UA_{M1}, UA_{M2} …, UA_{Mn}. Deux méthodes M_i et M_j sont directement reliées par la relation UA si $UA_{Mi} \cap UA_{Mj} \neq \emptyset$, c.-à-d. il y a au moins un attribut partagé (directement ou indirectement) par les deux méthodes.

Exemple 1 : Selon la figure II.2, les méthodes M_1 et M_2 sont directement reliées par la relation UA puisqu'elles partagent le même attribut A_1. M_1, M_3 et M_4 sont aussi reliées par la relation UA, car la méthode M_1 accède directement à l'attribut A_2, tout comme M_3, alors que M_4 appelle M_3 qui accède à A_2. En effet, $UA_{M1} = \{ A_1, A_2 \}$, $UA_{M2} = \{ A_1 \}$, $UA_{M3} = \{ A_2 \}$, et $UA_{M4} = \{A_2\}$; par conséquent, on aura: $UA_{M1} \cap UA_{M2} = \{A_1\}$, $UA_{M1} \cap UA_{M3} = \{A2\}$, et $UA_{M1} \cap UA_{M4} = \{A_2\}$.

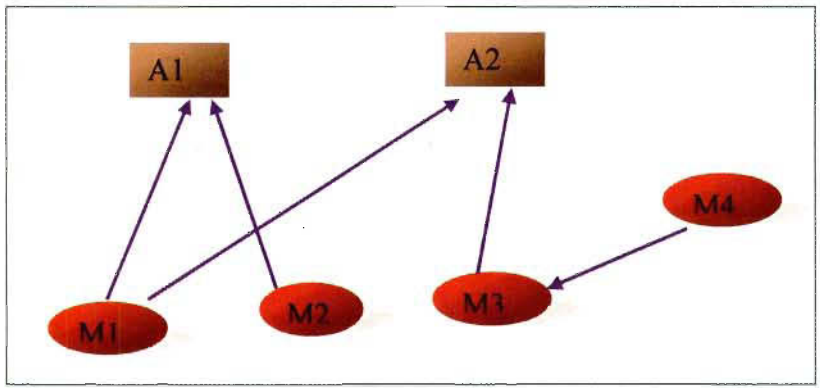

Figure II.2 : Relation d'usage d'attributs (UA).

II.2.2. Critère d'invocation de méthodes (IM)

Définition 2 [Badri 08]: Considérons une classe C. Soit $M = \{M_1, M_2, \ldots, M_n\}$ l'ensemble de ses méthodes. Soit IM_{Mi} l'ensemble de toutes les méthodes de la classe C qui sont invoquées directement ou indirectement par la méthode M_i. Une méthode M_j est appelée directement par une méthode M_i, si M_j apparaît dans le corps de M_i. Une méthode M_j est indirectement appelée par une méthode M_i si elle est appelée directement par une autre méthode de la classe C qui est invoquée directement ou indirectement par M_i. Il y a n ensembles IM_{M1}, IM_{M2}, …, IM_{Mn}. Deux méthodes M_i et M_j sont directement reliées par la relation IM si $IM_{Mi} \cap IM_{Mj} \neq \emptyset$, c.-à-d. il y a au moins une méthode conjointement partagée (directement ou indirectement) par les deux méthodes. On considère aussi que M_i et M_j sont directement reliées si $M_j \in IM_{Mi}$ ou $M_i \in IM_{Mj}$.

Exemple 2 : Selon la figure II.3, les méthodes M_1 et M_4 sont directement reliées par la relation IM car elles utilisent la même méthode M_2. Les méthodes M_1 et M_5 sont aussi reliées par la relation IM puisque M_1 utilise directement M_2 comme c'est le cas pour

M_4, qui est utilisée à son tour par la méthode M_5. En effet, $IM_{M1} = \{M_2\}$, $IM_{M4} = \{M_2, M_3\}$, et $IM_{M5} = \{M_2, M_3, M_4\}$. Par conséquent, on aura: $IM_{M1} \cap IM_{M4} = \{M_2\}$, $IM_{M1} \cap IM_{M5} = \{M_2\}$, et $IM_{M4} \cap IM_{M5} = \{M_2, M_3\}$.

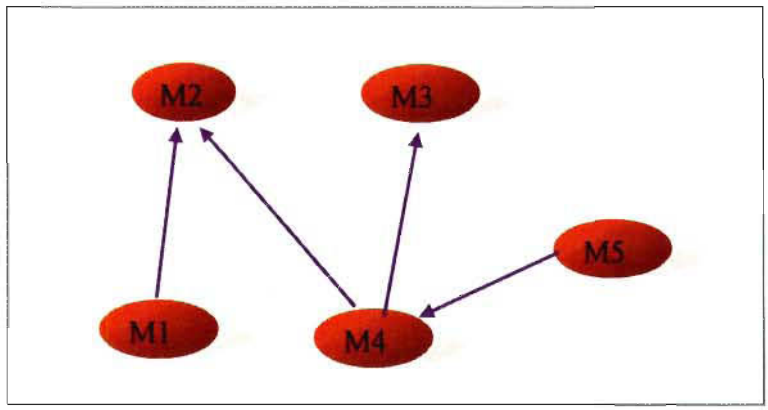

Figure II.3 : Relation d'invocation de méthodes (IM).

II.2.3. Critère arguments de type objet communs (CO)

Définition 3 [Badri 08] **:** On considère une classe C. Soit $M = \{M_1, M_2, \ldots, M_n\}$ l'ensemble de ses méthodes. Soit UCO_{Mi} l'ensemble de tous les paramètres (de type objet) d'une méthode M_i. Il y a n ensembles UCO_{M1}, UCO_{M2}, …, UCO_{Mn}. Deux méthodes M_i et M_j sont directement reliées par la relation UCO si $UCO_{Mi} \cap UCO_{Mj} \neq \emptyset$. (C.-à-d. il existe au moins un paramètre du même type objet utilisé par les deux méthodes).

II.2.4. Cohésion basée sur la relation directe

Définition 4 [Badri 08] **:** Deux méthodes publiques M_i et M_j peuvent être directement connectées de différentes manières : Elles partagent au moins une variable d'instance en

commun (relation UA : usage d'attributs), ou interagissent au moins avec une méthode de la même classe (relation IM invocation de méthodes), ou partagent au moins un objet passé comme argument (relation CO paramètres objets en commun). Donc, deux méthodes peuvent être directement connectées par un ou plusieurs critères.

➔ $UA_{Mi} \cap UA_{Mj} \neq \emptyset$ ou $IM_{Mi} \cap IM_{Mj} \neq \emptyset$ ou $UCO_{Mi} \cap UCO_{Mj} \neq \emptyset$.

Considérons un graphe non dirigé $\mathbf{G_D}$ où chaque nœud représente une méthode de la classe. Il y a un arc entre deux méthodes $\mathbf{M_i}$ et $\mathbf{M_j}$ si elles sont directement reliées. Soit $\mathbf{E_D}$ le nombre d'arcs dans le graphe $\mathbf{G_D}$.

Le degré de cohésion dans la classe C basé sur la relation directe entre ses méthodes publiques est défini par :

$$DC_{DE} = |E_D| / [n * (n - 1) / 2] \qquad (II.1)$$

$DC_{DE} \in [0,1]$.

Il donne le pourcentage de paires de méthodes publiques qui sont directement reliées. La métrique LCC_{DE} (Lack of Cohesion) dans la classe est alors donnée par :

$$LCC_{DE} = 1 - DC_{DE} \qquad (II.2)$$

$LCC_{DE} \in [0,1]$.

II.2.5. Cohésion basée sur la relation indirecte

Définition 5 [Badri 08] **:** Deux méthodes publiques M_i et M_j peuvent être indirectement reliées si elles sont directement ou indirectement reliées à une méthode M_k. Le concept

de fermeture transitive de la relation directe introduit par Bieman et Kang dans [Bieman 95] est appliqué pour identifier les méthodes reliées indirectement.

Considérons un graphe non orienté G_I, où les sommets sont les méthodes publiques d'une classe C. Il y a un arc entre deux sommets si les méthodes correspondantes sont directement ou indirectement reliées (la fermeture transitive du graphe G_D). Soit E_I le nombre d'arcs du graphe G_I. Alors, le degré de cohésion dans la classe C est défini par :

$$DC_{IE} = |E_I| / [n * (n - 1) / 2] \qquad (II.3)$$

$DC_{IE} \in [0,1]$.

DC_{IE} (extension de DC_I [Badri 04]) : donne le nombre de paires de méthodes publiques qui sont directement ou indirectement reliées. La valeur de la métrique de cohésion exprime le degré de liaison entre les membres d'une classe. Une valeur faible indique que les membres de la classe sont faiblement reliés entre eux.

Il faut noter que, dans la suite de cette analyse, c'est cette dernière métrique (DC_{IE}) que nous allons utiliser pour mesurer la cohésion des classes données en exemples. Elle sera considérée comme la métrique représentative de ce que nous référençons dans le texte comme ancienne approche d'évaluation de la cohésion de classe dans ce mémoire. Par ailleurs, elle sera aussi utilisée dans ce travail pour représenter les métriques de cohésion dites structurelles.

II.3. Limites des métriques de cohésion structurelles

Les métriques structurelles (dites aussi syntaxiques) sont basées sur la syntaxe du code et capturent la cohésion en termes de connexions entre les membres d'une classe en

comptant (par exemple) le nombre de variables d'instance utilisées par les méthodes ou les nombres de paires de méthodes qui partagent des variables d'instance. Elles capturent le degré avec lequel les éléments d'une classe sont liés ensemble du point de vue structurel.

Bien que les efforts des chercheurs se multiplient dans le but d'améliorer l'évaluation de la cohésion des classes dans les systèmes orientés-objet, celle-ci demeure un attribut vraiment complexe et difficile à capturer. En effet, ces métriques ont plusieurs limites et manquent souvent (pour certaines d'entre elles) d'argumentation.

Plusieurs études ont noté, en effet, qu'elles échouent dans plusieurs situations à refléter proprement la cohésion d'une classe [Kabaili 00, Chae 00, Aman 02]. Selon plusieurs auteurs, elles ne tiennent pas compte de certaines caractéristiques des classes, par exemple : la taille des parties cohésives [Aman 02] et la connectivité parmi les membres [Chae 00]. Aussi, les métriques LCOM n'aident pas à distinguer les classes partiellement cohésives [Bieman 95]. L'autre problème, avec les différentes métriques proposées par Chidamber et Kemerer et les autres métriques LCOM, consiste dans le fait qu'elles aident seulement à identifier les classes non-cohésives. En effet, elles mesurent seulement l'absence de cohésion au lieu de la présence de cohésion [Etzkorn 03]. Par ailleurs, selon [Etzkorn 03] TCC et LCC ne conviennent pas vraiment pour mesurer la cohésion d'une classe ayant un nombre élevé de méthodes publiques (à cause de leur dénominateur). Marcus mentionne que la cohésion est généralement mesurée seulement sur la base de l'information structurelle extraite du code source (par exemple, les attributs référencés par les méthodes et les appels de méthodes) qui capture le degré avec lequel les éléments d'une classe sont reliés ensemble du point de vue structurel [Marcus 08]. Le principe derrière cette catégorie de métriques est de mesurer le couplage entre les méthodes d'une classe. Par conséquent, elles ne donnent que peu d'information à propos de la cohésion du point de vue conceptuel (par exemple, quand

une classe implémente un ou plusieurs concepts de domaine). Elles ne donnent pas également selon [Marcus 08] d'indication sur la facilité de lecture ou de compréhension du code source.

En effet, il y a d'autres aspects de la cohésion d'une classe autre que structurels. Il s'agit de la cohésion conceptuelle (ou sémantique). Henderson-Sellers mentionne : « Après tout, il est possible d'avoir une classe ayant une cohésion syntaxique interne élevée, mais d'une faible cohésion sémantique » [Henderson-Sellers 96]. Les métriques basées seulement sur la syntaxe ne capturent pas bien la cohésion sémantique. Elles ne peuvent donc pas capturer tous les aspects et caractéristiques de la cohésion d'une classe. La cohésion conceptuelle d'une classe capture une nouvelle dimension complémentaire de la cohésion comparée aux métriques structurelles existantes [Marcus 08]. Etzkorn et al. prétendent qu'une métrique de cohésion qui pourrait être préférée par les développeurs serait une métrique qui reflète le mieux une vue orientée « humain » de la cohésion [Etzkorn 03]. Une cohésion orientée humain signifie en gros: collecter les tâches dans des classes quand elles sont logiquement reliées du point de vue humain, au lieu de regrouper les segments de code qui sont syntaxiquement reliés, mais peuvent être dans certains cas non reliés logiquement [Etzkorn 03].

La cohésion conceptuelle des classes capture les aspects conceptuels de la cohésion en mesurant comment les méthodes d'une classe sont fortement reliées les unes aux autres conceptuellement. La relation conceptuelle entre les méthodes est basée sur le principe de cohérence textuelle selon [Marcus 08]. Elle est basée, dans certaines approches, sur l'analyse de l'information textuelle dans le code source, exprimée dans les commentaires et les identificateurs. Les métriques sémantiques quantifient la sémantique de la tâche exécutée par une pièce du logiciel, sur la base d'informations relatives aux concepts et des mots-clés du domaine relié. Elles mettent l'accent sur le sens ou la signification d'un logiciel au sein du domaine de problème. Enfin, une chose

est certaine, il n'y a pas encore une métrique de cohésion satisfaisante ou qui est admise du moins majoritairement! Ceci en raison de plusieurs causes comme celles que nous venons de citer.

Nous croyons que le problème revient plutôt à trouver une approche appropriée permettant de combiner les différents critères de cohésion afin de concevoir une mesure capable de capturer les aspects structurels et surtout conceptuels liés à la cohésion d'une classe. La classification représente un moyen naturel à notre avis pour combiner ces critères de façon appropriée et capturer les différents liens conceptuels possibles entre les membres d'une classe.

Il nous semble aussi que certaines métriques surestiment la cohésion d'une classe en termes de connectivité entre les méthodes. Certes deux méthodes peuvent être directement ou indirectement reliées structurellement, mais en fait, est-ce qu'elles le sont aussi conceptuellement? Ceci n'étant pas toujours vrai, est-ce qu'il y a un moyen pour déterminer les connectivités conceptuelles entre les méthodes sur la base des connexions de bas niveau ou structurelles. La classification est une technique qui s'intègre justement dans cette problématique. C'est une technique qui a le potentiel de survoler les détails des éléments et des propriétés de bas niveau, et de les abstraire afin de détecter les concepts qu'ils caractérisent à un niveau supérieur. Nous nous proposons d'explorer ce potentiel dans le présent travail.

CHAPITRE III

LA CLASSIFICATION

III.1. Introduction

La classification est une des techniques de forage de données (data mining) qui a pour but de différencier des groupes d'objets (clusters) au sein d'un ensemble d'objets donné [Han 01]. L'analyse de classification est le nom générique d'une grande variété de procédures qui peuvent être utilisées pour former de façon empirique des groupes d'entités fortement similaires (ou clusters). Plus spécifiquement, une méthode de classification est une procédure statistique multi variée qui commence par un ensemble de données contenant l'information au sujet d'un échantillon d'entités et essaie de réorganiser ces entités en des groupes relativement homogènes [Aldenderfer 85].

La classification peut être de deux types : classification supervisée et classification non supervisée. La classification supervisée vise à classer des objets selon des catégories bien définies au préalable. Dans la classification non supervisée, les classes ne sont pas connues à l'avance et doivent ressortir comme résultat lors du processus de la classification selon des critères de similarité choisis.

Les techniques de classification ont été utilisées avec succès dans plusieurs domaines pour assister au regroupement des composants similaires et supporter le partitionnement des systèmes. Elles peuvent faciliter une meilleure compréhension des observations et la construction subséquente de structures complexes de connaissances à partir des propriétés et des clusters composantes [Lung 04c]. Elles ont été utilisées dans différents domaines

pour traiter un large spectre de problèmes parmi lesquels on peut citer : la théorie des graphes, l'architecture de l'information, la recherche et l'extraction de l'information, l'allocation des ressources, le traitement d'images, le test des logiciels, la reconnaissance de modèles (patterns), les statistiques et la biologie [Wiggerts 97]. La classification des documents, par exemple, permet de regrouper les documents similaires dans un même ensemble dans le but d'accélérer le processus de recherche et de regrouper les résultats.

Le concept clé de la classification est de regrouper les choses similaires ensemble, pour former un ensemble de clusters de sorte à ce que la similarité intra cluster (cohésion) soit élevée et la similarité inter clusters (couplage) soit faible. L'objectif – forte cohésion et faible couplage – est semblable à celui de la conception des logiciels. En fait, la classification des composants a été largement discutée dans la littérature du génie logiciel pour supporter la restructuration des systèmes et la réingénierie [Lung 04b, Lethbridge 99, Lung 98, Snelting 00, Lung 04c, Wiggerts 97]. Elle a été appliquée aussi pour différentes raisons à différents niveaux d'abstraction et dans différentes phases du cycle de développement. On l'a utilisée pour supporter le partitionnement du système durant la phase de conception [Lung 04c], pour le regroupement des composants basé sur le code source pour la récupération de l'architecture logicielle «architecture recovery» lors d'un processus d'ingénierie inverse [Lung 04c], dans la restructuration des programmes afin de supporter l'évolution dans la phase de maintenance [Lung 04b, Czibula 07d], et aussi dans l'amélioration de la cohésion et la réduction du couplage du code source.

III.2. Notion de similarité et de dissimilarité

La classification consiste à séparer un ensemble d'objets en différents groupes (ou clusters) en fonction d'une certaine notion de similarité. Les objets considérés comme

similaires sont ainsi associés au même cluster, alors que ceux qui sont considérés comme différents sont associés à des clusters distincts. L'objectif étant de définir des groupes d'objets de sorte que la similarité entre objets d'un même groupe soit maximale et que la similarité entre objets de groupes différents soit minimale. Par conséquent, le choix ou la définition d'une mesure de similarité appropriée entre objets est fondamental. La mesure de la proximité entre deux objets peut se faire en mesurant à quel point ils sont semblables ou dissemblables en faisant appel aux mesures de similarité.

Soit E l'ensemble des objets à classer. Une mesure de dissimilarité d est une application de $E \times E$ dans R+ telle que :

$$d(i,i) = 0 \; ; \; \forall i \in E \tag{III.1}$$

$$d(i,i') = d(i',i) \; ; \; \forall i,i' \in E \times E \tag{III.2}$$

Il faut noter que la distance est un cas particulier des mesures de dissimilarité. En effet, pour que d soit une distance, il faut qu'en plus des propriétés (III.1) et (III.2), l'inégalité triangulaire (III.3) soit satisfaite aussi.

$$d(i,j) \leq d(i,r) + d(r,j); \; \forall i,j,r \in E \times E \times E \tag{III.3}$$

La **matrice des similarités** (ou matrice des proximités) décrit la matrice N × N des similarités entre les observations après avoir soumis les données de la matrice à une certaine mesure de similarité ou de dissimilarité. Le terme similarité a plusieurs synonymes qui sont : « ressemblance », « proximité » et « association ».

III.3. Mesures de similarité

Elles permettent de mesurer la similarité entre deux entités. Le choix d'une mesure de similarité aura une grande influence sur le résultat de la classification [Lethbridge 03]. Il existe plusieurs mesures de similarité classées en quatre catégories selon la classification établie par Sneath [Sneath 73], et reprise par Wiggerts [Wiggerts 97]:
(1) les coefficients de corrélation, (2) les mesures de distance, (3) les coefficients d'association, et (4) les mesures de similarité probabiliste.

Nous donnons, dans ce qui suit, une brève description de chaque catégorie telle que reportée par Aldenderfer [Aldenderfer 85].

III.3.1. Coefficients de corrélation

Ils ont une interprétation géométrique (mesures angulaires). Le plus populaire est le coefficient de corrélation produit-moment proposé par Karl Pearson, qui est défini à l'origine comme une méthode pour corréler les variables. Il a été utilisé dans la classification quantitative pour déterminer la corrélation entre les cas. Dans ce contexte, ce coefficient est donné par la formule suivante :

$$r_{ij} = \frac{\sum(x_{ij}-\bar{x}_j)(x_{ik}-\bar{x}_k)}{\sqrt{\sum(x_{ij}-\bar{x}_j)^2 \sum(x_{ik}-\bar{x}_k)^2}} \qquad \text{(III.4)}$$

Où

x_{ij} : est la valeur de la variable i pour le cas j,

$\bar{x}_j$: est la moyenne de toutes les valeurs de la variable pour le cas j.

La valeur du coefficient est comprise entre -1 et +1. La valeur 0 indique qu'il n'y a aucune relation entre les cas.

III.3.2. Mesures de distance

Elles sont mieux décrites comme étant des mesures de dissimilarité (inversement aux coefficients de similarité). Deux cas sont identiques si chacun est décrit par des variables avec la même magnitude, et donc la distance entre eux est nulle [Aldenderfer 85].

- La ***distance euclidienne*** étant la plus populaire. Elle est définie par :

$$d_{ij} = \sqrt{\sum_{k=1}^{p}\left(x_{ik} - x_{jk}\right)^2} \qquad \text{(III.5)}$$

- La ***distance de Manhattan*** est définie par :

$$d_{ij} = \sum_{k=1}^{p}\left|x_{ik} - x_{jk}\right| \qquad \text{(III.6)}$$

Où

d_{ij} : est la distance entre deux cas i et j.

x_{ik} : est la valeur de la k^{eme} variable pour le i^{eme} cas.

- D'autres métriques sont des cas spécifiques (incluant les deux précédentes) de la classe des fonctions de métriques de distance connues sous le nom de ***métrique de Minkowski***, définie comme suit :

$$d_{ij} = \left(\sum_{k=1}^{p} \left|x_{ik} - x_{jk}\right|^{r}\right)^{1/r} \qquad \text{(III.7)}$$

III.3.3. Coefficients d'association

Ils sont utilisés pour déterminer la similarité entre les cas décrits par des variables binaires. Il est facile d'expliquer ces coefficients par référence à la table d'association 2X2 (Tableau III.1), dans laquelle 1 indique la présence de la variable et 0 son absence [Aldenderfer 85].

		Variable Y	
		1	*0*
Variable X	*1*	**a**	**b**
	0	**c**	**d**

Tableau III.1 : Table d'association 2X2.

La case « a » du tableau précédent représente le nombre de cas où les deux variables X et Y partagent une même propriété. Les cases « b » et « c » sont le nombre de cas où une seule des deux variables possède la propriété. Il faut noter que (a+b+c+d =n) représente la taille de l'ensemble des propriétés. Ainsi, la similarité entre deux entités X et Y est exprimée en utilisant les quatre termes précédents qui peuvent être décrits formellement comme suit :

$$a = \|X \cap Y\| \quad , b = \|X \backslash Y\| \quad , c = \|Y \backslash X\| \quad , \; d = \|P \backslash (X \cup Y)\|$$

Où P est l'ensemble de toutes les propriétés.

Les coefficients d'associations les plus populaires sont le coefficient simple, et le coefficient de Jaccard.

- ***Coefficient Simple*** appelé aussi indice de Sokal & Michener) est défini par :

$$S_S = (a+d)/ (a + b + c+d) \qquad \text{(III.8)}$$

Coefficient Jaccard défini par Aldenderfer [Aldenderfer 85] comme suit:

$$S_J = a/ (a + b + c) \qquad \text{(III.9)}$$

Il évite l'utilisation des variables absentes dans le calcul de la similarité (il ignore les cellules d). Si S est la similarité entre deux cas, S appartient à l'intervalle [0,1]. Il est possible de l'écrire aussi sous la forme suivante :

$$S_J = \frac{\|X \cap Y\|}{\|X \cup Y\|} \qquad \text{(III.10)}$$

Puisque $\|X \cup Y\| = \|X \cap Y\| + \|X \backslash Y\| + \|Y \backslash X\|$

III.3.4. Coefficient de similarité probabiliste

Il est utilisé seulement avec des données binaires. Techniquement différent des précédents, la similarité entre deux cas n'est pas réellement calculée [Aldenderfer 85].

En revanche, ce type de mesure travaille directement sur les données brutes de la matrice de données initiale.

Le choix des indices de similarité dépend de la nature des données à classifier qui peut être quantitative, qualitative ou binaire. Par exemple, parmi les indices de similarité proposés pour des calculs à partir de données binaires, on trouve : l'indice de Dice (de Sorensen), l'indice de Jaccard, l'indice de Kulczinski, Phi de Pearson, l'indice d'Ochiai, l'indice de Rogers & Tanimoto, l'indice de Sokal & Michener (coefficient simple) et les indices de Sokal & Sneath.

D'ailleurs, dans la plupart des travaux menés dans [Czibula 07b], [Czibula 07c], [Czibula 07d], [Czibula 08] et [Czibula 07a] dans le domaine du refactoring et aussi de l'aspect mining, les auteurs ont utilisé une mesure absolument équivalente ou presque au coefficient de Jaccard, mais formulée et désignée différemment (quotient de l'intersection des propriétés communes sur l'union des propriétés). Ce choix n'est pas aléatoire, puisque les matrices de données (ou entités-propriétés) sont purement binaires. Il serait plus judicieux d'opter pour un coefficient d'association comme mesure de similarité, plutôt que toute autre mesure de distance pour être conforme à la nature des données traitées. C'est pourquoi notre choix s'est fixé aussi sur le coefficient de Jaccard vu que nous avons à faire à des données binaires.

III.4. Méthodes de la classification

On distingue deux grandes familles de méthodes : **méthodes par partition et méthodes hiérarchiques.**

III.4.1. Les méthodes par partition [Candillier 06]

Le résultat de ce type de méthode fournit une partition de l'espace des objets, c.-à-d. que chaque objet est associé à un unique cluster. Il y a plusieurs méthodes :

a) La classification statistique
b) La classification stochastique
c) La classification basée sur les grilles
d) La classification basée sur les graphes
e) La classification spectrale

La fameuse **méthode K-means** qui a été proposée par Diday et al. [Diday 82], est un cas particulier de la première famille (c.-à-d. la classification statistique). Il serait convenable d'en donner dans ce qui suit une brève description:

Étant donné le nombre K de clusters recherchés (connu à l'avance) :

1- Choisir aléatoirement k objets parmi l'ensemble d'objets initial pour former des « centroïdes » initiaux représentant les k clusters recherchés.
2- Assigner chaque objet au cluster dont le « centroïde» est le plus proche.
3- Tant qu'au moins un objet change de clusters d'une itération à l'autre :
   a. Mettre à jour les centroïdes des clusters en fonction des objets qui leur sont associés.
   b. Mettre à jour les assignations des objets aux clusters en fonction de leurs proximités aux nouveaux centroïdes[1].

[1] **Centroide**: est un terme à usage général selon le contexte (géométrie, physique, ...). C'est une moyenne de tous les points composants, pondérée par des poids spécifiques respectifs. Par exemple en physique le centroide désigne le centre géométrique d'une forme géométrique, il peut aussi désigner le centre de gravité d'une masse. Par analogie, en classification, le centroide d'un cluster représente le centre de masse de ce dernier [Romesburg 84].

III.4.2. Les méthodes hiérarchiques [Candillier 06]

Ici, une hiérarchie de clusters est formée de telle manière que plus on descend dans la hiérarchie, plus les clusters sont spécifiques à un certain nombre d'objets considérés comme similaires. Il y a deux types de méthodes:

a) Méthodes ascendantes

Elles démarrent avec autant de clusters que d'objets initiaux à classifier, puis fusionnent successivement les clusters considérés comme les plus similaires (selon un certain critère) jusqu'à ce que tous les objets soient finalement regroupés dans un unique cluster. En travaillant à partir des dissimilarités, l'algorithme procède selon les étapes suivantes :

- On commence par calculer la dissimilarité entre les N objets à classifier.

- Puis, à chaque itération, on regroupe les deux objets ou classes dont le regroupement minimise un certain critère d'agrégation donné, créant ainsi un nouveau cluster comprenant ces deux derniers. On recalcule ensuite la dissimilarité entre ce nouveau cluster et les autres objets restants selon le critère d'agrégation. On continue ainsi jusqu'à ce que tous les objets soient regroupés.

b) Méthodes descendantes

Elles démarrent avec un unique cluster regroupant l'ensemble des objets à classifier, puis divisent successivement les clusters de manière à ce que les clusters résultants

soient les plus différents possible et ce jusqu'à obtenir au bas de la hiérarchie autant de clusters que d'objets initiaux.

Dans les deux types de méthodes : étant donnée une mesure de similarité (ou dissimilarité) entre deux clusters, la fusion suivante à effectuer pour une méthode ascendante concernera les deux clusters les plus similaires (proches au sens de distance), alors que la division suivante à effectuer pour une méthode descendante concernera les deux clusters les plus dissemblables (éloignés).

Elles produisent un arbre binaire de classification (appelé dendrogramme), qui permet de visualiser le regroupement progressif des données. On peut alors se faire une idée d'un nombre adéquat de classes dans lesquelles les données peuvent être regroupées. Ce dendrogramme représente une hiérarchie de partitions. On peut alors choisir une partition en tronquant l'arbre à un niveau donné, ce dernier dépendant soit des contraintes de l'utilisateur (l'utilisateur sait combien de classes il veut obtenir), soit de critères plus objectifs (on peut spécifier des seuils sur la cohésion interne ou l'isolation externe minimale des clusters).

c) Méthodes d'agrégation

Il est nécessaire de définir une méthode d'agrégation entre un objet et un groupe d'objets (cluster) ou entre deux groupes d'objets (deux clusters). Pour calculer la dissimilarité, notée δ, entre deux groupes d'objets A et B, différentes stratégies sont possibles parmi lesquelles on peut citer :

- Lien simple (single-link) : $\delta(A, B) = min \{d(a, b), a \in A, b \in B\}$
- Lien complet (complete-link) : $\delta(A, B) = max \{d(a, b), a \in A, b \in B\}$
- Lien moyen (average-link) : $\delta(A, B) = moy \{d(a, b), a \in A, b \in B\}$

- Lien proportionnel
- Lien flexible
- Méthode de Ward

Il y a aussi différentes méthodes hybrides qui ont été proposées en combinant les caractéristiques des méthodes hiérarchiques et des méthodes par partitions cherchant ainsi à profiter des atouts de chaque méthode.

III.5. Détermination du nombre optimal de clusters (partitions)

Le critère de troncature de l'arbre hiérarchique est souvent basé sur des heuristiques [Valdano 03]. Il y a plusieurs méthodes pour déterminer le nombre optimal de clusters. Des exemples de travaux anciens sont :

- La règle de Hartigan [Hartigan 75],
- Les indices de Krzanowski and Lai [Krzanowski 88],
- La statistique silhouette suggérée par Kaufman et al. [Kaufman 90].

D'autres travaux plus récents dans ce domaine incluent:

- La méthode « gap » proposée par Tibshirani et al [Tibshirani 01],
- La méthode de ré-échantillonnage basée sur la prédiction [Dudoit 02].

Afin d'avoir une idée sur le principe derrière ces techniques, nous allons décrire brièvement dans ce qui suit la base d'une des plus importantes d'entre elles. Il s'agit de la méthode (Scree plots).

- **La méthode du « Scree plots »**

Elle est décrite dans [Fielding 07] par l'exemple ci-après. Selon le dendrogramme de la figure III.1, les données peuvent être représentées par deux clusters : A (cas 1 et 2) et B (cas : 3,4 et 5). Cependant, quand le nombre de cas augmente, il ne peut pas être si évident de distinguer entre les clusters et l'évaluation visuelle du dendrogramme deviendrait très subjective.

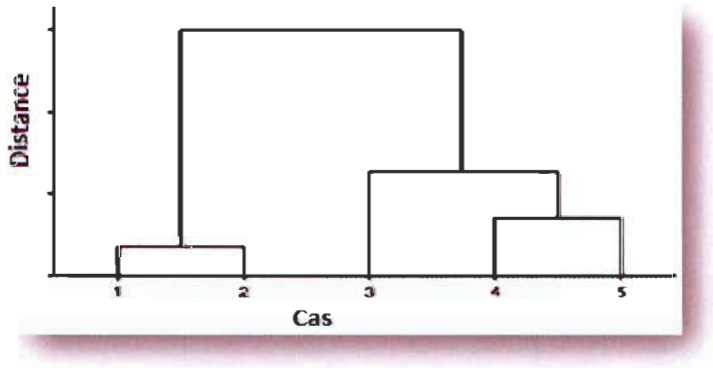

Figure III.1 : Arbre hiérarchique identifiant deux clusters majeurs *[Fielding 07]*.

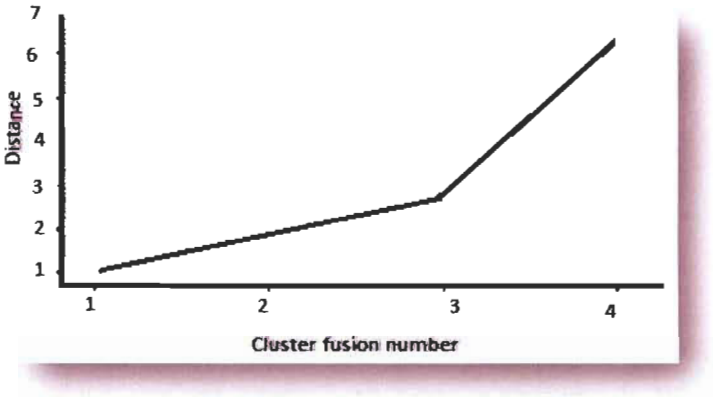

Figure III.2 : La courbe « Scree plot » basée sur l'analyse de la figure III.1 *[Fielding 07]*.

Dans une analyse hiérarchique, les clusters graduellement dissemblables devraient être fusionnés tant que le processus de fusion de clusters continue. Par conséquent, il est probable que la classification devienne graduellement artificielle. Un graphe (courbe) des niveaux de similarité, à la fusion versus le nombre de clusters, peut aider à reconnaitre les points auxquels certains clusters deviennent artificiels parce qu'il y aura des sauts brusques dans les nivaux de similarité quand des groupes dissemblables sont fusionnés. La figure III.2 dérive de l'analyse précédente. On note comment il y a eu un grand saut quand les clusters A et B sont fusionnés. Ceci supporte l'hypothèse que les données sont mieux représentées par deux clusters.

Notion d'entropie en théorie de l'information

L'entropie fait partie des mesures pour évaluer la qualité d'une classification. Ces mesures sont déduites des mesures d'évaluation des systèmes de recherche d'informations telles que le rappel et la précision. L'entropie mesure la façon dont les objets sont distribués ou répartis dans chaque cluster. Une valeur faible de l'entropie correspond à une meilleure classification.

Soit un cluster S_r de taille n_r. L'entropie de ce cluster est définie comme suit :

$$E(s_r) = -\frac{1}{\log(q)} \sum_{i=1}^{q} \frac{n_r^i}{n_r} \log \frac{n_r^i}{n_r} \qquad \text{(III.11)}$$

q : Nombre de classes.

n^i_r : Nombre de documents dans la classe i, affectés au cluster r.

n_r : Nombre de documents du cluster r.

La formule générale de l'entropie est :

$$E = \sum_{r=1}^{k} \frac{n_r}{n} E(s_r) \qquad \text{(III.12)}$$

k **:** Nombre de clusters.

n **:** Nombre de documents de la collection.

L'outil XlStat de Addinsoft se base sur l'entropie pour déterminer le nombre optimal de clusters (La méthode n'est cependant pas décrite dans l'aide en ligne du logiciel). Étant donnée la grande simplicité de cet outil, son efficacité ainsi que la disponibilité d'une fonction de troncature automatique de l'arbre hiérarchique, c'est ce dernier qui a été retenu pour réaliser nos expérimentations. C'est aussi une des raisons pour lesquelles nous avons opté pour la méthode hiérarchique ascendante. D'ailleurs, une étude comparative sur plusieurs algorithmes de classification dans le cadre de l'aspect mining, Czibula et al. conclut que la classification hiérarchique semble plus appropriée dans l'aspect mining que les méthodes de partitionnement [Czibula 07a]. Toutefois, on aurait pu choisir la méthode par partitions, cela pourrait faire (ultérieurement) l'objet d'une éventuelle démarche de comparaison entre les deux approches par exemple. L'idée principale dans ce projet étant d'abord d'explorer l'utilisation de la classification pour évaluer la cohésion des classes.

III.6. Conclusion

La classification aide à découvrir la structuration des données qui n'est pas facilement apparente par inspection visuelle [Aldenderfer 85]. Elle peut faciliter une meilleure compréhension des observations et la construction subséquente de structures complexes

de connaissances à partir des propriétés et des clusters composantes [Lung 04c]. Le principe de la classification consiste à regrouper les éléments similaires ensemble pour former un ensemble de clusters de sorte à maximiser la similarité au sein d'un même cluster, à maximiser la cohésion interne au sein d'un même cluster, à faire en sorte que la similarité inter clusters soit faible, et enfin à minimiser le plus possible le couplage entre les différents clusters. On cherche une meilleure isolation des clusters tout en maximisant la cohésion de chaque cluster.

Cette technique offre un double avantage. Elle permet d'une part, de découvrir les regroupements cohésifs possibles au sein d'un ensemble, et d'autre part, de regrouper ces éléments cohésifs au sein d'un même cluster. Nous pensons que l'approche de classification va nous aider à déterminer les éléments cohésifs au sein d'une classe et à les classifier selon leur réelle similarité de façon plus appropriée. Nous pensons également que cela va mieux nous aider à déceler d'éventuelle disparité dans le code source.

CHAPITRE IV

TRAVAUX CONNEXES RELATIFS À LA CLASSIFICATION

La classification a été appliquée dans plusieurs domaines tels que la réingénierie, la restructuration des programmes ainsi que l'aspect mining. Plusieurs travaux antérieurs sur la classification des logiciels tels que [Wiggerts 97, Hutchens 85, Lung 04b, Lung 04c, Czibula 06, Czibula 07b, Czibula 07c, Czibula 07d, Czibula 08], pour n'en citer que quelques uns, concluent que la classification est une technique efficace pour la restructuration des logiciels. Dans ce qui suit, nous allons présenter brièvement certains de ces travaux.

IV.1. La classification dans le contexte de la restructuration des programmes

Notre approche s'est inspirée surtout du travail de Simon et al. [Simon 01]. Ils ont montré comment identifier les parties d'un système où certaines opérations de restructuration concernant les membres d'une classe sont nécessaires. Pour mesurer le degré avec lequel certaines parties sont reliées ensemble, les auteurs se sont basés sur une mesure de cohésion générique fortement liée à la théorie de la similarité (classification). Ainsi, la similarité entre deux entités dépend de la collection des propriétés partagées. Sur la base d'un ensemble B de propriétés considérées d'un point de vue spécial de similarité, les auteurs ont défini une mesure de distance entre deux entités x et y par :

$$dist(x,y) = 1 - \frac{|\, p(x) \cap p(y) \,|}{|\, p(x) \cup p(y) \,|} \qquad \text{(IV.1)}$$

$$p(x) := \{ propriétés \ \ p_i \in B \mid x \ \ possede \ \ p_i \} \qquad \text{(IV.2)}$$

Selon les auteurs, cette approche supporte la définition de la cohésion. Ainsi, les parties qui sont à faible distance sont cohésives, tandis que celles à grande distance le sont moins. Il est facile de noter que cette mesure dist(x,y) est équivalente à la définition du coefficient de Jaccard, défini un peu plus loin dans la section III.3.3. D'ailleurs c'est cette mesure qui constitue la base des travaux de plusieurs autres chercheurs tels que [Czibula 07a], [Czibula 07b], [Czibula 07c], [Czibula 07d], et [Czibula 08].

Dans une classe, une méthode peut utiliser des méthodes et des attributs. L'ensemble des entités est formé par les attributs et les méthodes. Ainsi, pour une méthode, l'ensemble de ses propriétés est formé par : la méthode elle-même, toutes les méthodes directement utilisées et tous les attributs directement utilisés. Pour un attribut donné, l'ensemble des propriétés est formé par : l'attribut lui-même, et toutes les méthodes qui l'utilisent.

Ici, la portée des propriétés n'est pas limitée à la classe où l'entité considérée a été définie, puisqu'elle touche toutes les classes du système. Contrairement à la nouvelle approche que nous avons adoptée (voir ultérieurement dans le mémoire), où la portée des propriétés est limitée à la classe. Ceci rejoint parfaitement le principe de la cohésion des composants logiciels (dépendances inter membres à l'intérieur du composant).

Pour calculer la distance entre deux membres m et a, l'ensemble de propriétés ***B*** est donné par l'union $B_m \cup B_a$. Par la suite, les distances entre chaque paire d'entités (attribut ou méthode) sont calculées. Le résultat est stocké dans une matrice des distances (qui équivaut à la matrice des proximités dans une démarche de classification).

Donc, une méthode utilisant plusieurs attributs ou méthodes définis dans d'autres classes (selon Simon et al.) a une courte distance vers ces membres, tandis qu'une méthode utilisant seulement les méthodes ou les attributs définis localement (dans une classe) a une large distance vers eux.

Il est clair que Simon et al. [Simon 01] ont implicitement utilisé la classification dans leur démarche pour identifier les opportunités de restructuration des classes, mais d'une manière différente de l'approche de classification classique. Ils se basent, d'ailleurs, sur la matrice des proximités (ou dissimilarités). Cependant, au lieu de continuer le traitement avec un algorithme de classification, ils ont plutôt opté pour un outil de visualisation permettant de visualiser la structure des classes de façon géométrique en se basant sur les distances euclidiennes équivalentes correspondantes (VRML : Virtual Reality Modeling Language).

Les recherches de Lung [Lung 04c] présentent une approche basée sur la classification dont le but est de réduire la complexité des composants (procédures ou fonctions) complexes par le découplage de ces derniers en des modules plus cohésifs. Deux variables d'output sont cohésives si elles dépendent d'au moins une variable d'input commune. L'identification des variables très reliées est semblable à la classification des variables en clusters cohésifs. Dans ce type d'application, chaque variable peut être traitée comme un composant. Les relations entre les variables d'input et d'output sont leurs interdépendances. Le but est de réduire la complexité des composants complexes.

Dans le cadre d'un processus de reverse engineering, Hutchens et Basili [Hutchens 85] ont spécifiquement appliqué la classification avec le « data binding » pour la « modularisation » d'un système afin de regrouper les procédures reliées logiquement. Le data binding est utilisé comme critère pour déterminer comment deux procédures sont reliées. Le nombre de data binding de flux de contrôle entre deux procédures est

utilisé pour calculer la similarité entre chaque paire de procédures. Plusieurs méthodes pour ce calcul sont présentées. L'une d'elles est le coefficient de Jaccard. La matrice résultante servira comme input pour l'algorithme de classification hiérarchique.

Dans un autre travail de Lung [Lung 04b], l'accent est mis sur comment la technique de classification peut être utilisée pour fournir une assistance à l'activité de restructuration en des fonctions cohésives au sein d'un programme. Les données d'entrée sont obtenues par le « parsing » du programme au niveau fonction, et le « mapping » des instructions logiques dans chaque fonction. La relation de dépendance d'une instruction est obtenue à partir de ses variables et de la portée au sein d'un module. Les instructions sont vues comme étant les composants à classifier et les variables sont considérées comme les propriétés. La portée des variables (If, Else, Loop) est aussi prise en compte et est traitée comme des variables, mais avec de faibles poids. L'ensemble des données en entrée obtenu forme alors une matrice des instructions – variables/portée. Dans cette étude, les auteurs [Lung 04b] ont choisi différents types d'associations avec différents poids pour calculer les coefficients. Pour calculer la similarité entre deux composants (instructions), ils calculent la proportion des correspondances appropriées entre celles-ci. Il y a plusieurs méthodes pour compter les correspondances (matches) et plusieurs algorithmes pour calculer les similarités ou le coefficient de ressemblance. Certaines heuristiques sont présentées pour le choix d'un algorithme particulier [Lung 04b].

Les coefficients vont en baisse au fur et à mesure que le processus progresse. Une chute considérable pourrait être une indication du fait qu'il y a des fonctionnalités multiples à l'intérieur d'une seule fonction. Le processus créera plusieurs groupes d'instructions où chaque groupe d'instructions pourrait être mappé vers une nouvelle fonction [Lung 04b].

Czibula présente une étude dans laquelle la classification hiérarchique ascendante a été utilisée dans le but d'améliorer la conception du système logiciel et réorganiser autrement la structure de ses classes [Czibula 07b]. Les membres des classes du logiciel existant (méthodes et attributs) pourraient être donc réassemblés à l'aide de la classification hiérarchique en suggérant des actions de « Refactoring » appropriées. La mesure de similarité utilisée est donnée par la formule (IV.3) [Czibula 07b].

$$d(s_i, s_j) = \begin{cases} 1 - \dfrac{| p(s_i) \cap p(s_j) |}{| p(s_i) \cup p(s_j) |} & si \;\; p(s_i) \cap p(s_j) \neq \Phi \\ \infty & \text{sinon} \end{cases} \qquad \text{(IV.3)}$$

Où $P(s_i)$ est l'ensemble des propriétés que possède l'entité s_i.

L'heuristique utilisée est telle que : à une étape donnée, les deux clusters les plus similaires sont fusionnés seulement si la distance entre eux est inférieure ou égale à un certain seuil donné distMin. Cela signifie que les entités des deux clusters sont assez proches pour les placer dans un même cluster. Dans cette approche, l'auteur a choisi la valeur 1 pour le seuil distMin parce qu'une distance supérieure à 1 correspond à des entités non reliées.

Une autre approche a été présentée par Czibula [Czibula 08] où elle propose un algorithme de classification par partitions afin d'améliorer la structure des systèmes logiciels orientés objet. L'approche procède comme suit : le système logiciel est analysé afin d'en extraire les entités : classes, méthodes, attributs et les relations existantes entre elles : relations d'héritage, d'agrégation, et de dépendances entre les entités. Ensuite, l'ensemble des entités extraites dans l'étape précédente, sont regroupées dans des

clusters (classes) à l'aide de l'algorithme de classification (PARED). Le but étant d'obtenir une structure améliorée du système existant. Enfin, la nouvelle structure obtenue du système est comparée avec la structure d'origine dans le but de fournir une liste d'opérations de « Refactoring » qui pourraient améliorer la structure d'origine. L'algorithme (K meldois) PARED utilise aussi une heuristique pour choisir à la fois le nombre de medoïdes (clusters) et les medoïdes initiaux. Après avoir sélectionné les medoïdes initiaux, PARED se comporte comme un algorithme k-Meldois classique.

Certains auteurs [Serban 08a] ont présenté une approche de détection des « Refactoring » automatiques basée sur la classification hiérarchique. Les objets à classifier sont les entités qui composent le système : les classes, les méthodes et les attributs. L'idée est de regrouper les entités similaires du système dans le but d'obtenir des groupes à forte cohésion. Serban et al. [Serban 08a] ont introduit deux algorithmes de classification hiérarchique ascendants qui ont pour but d'identifier une partition du système qui correspond à une structure améliorée de ce dernier. Les deux algorithmes utilisent une mesure pour l'évaluation de la partition du système du point de vue conception. Ils ont utilisé une heuristique pour fusionner deux clusters dans le processus ascendant, et une autre heuristique pour la détermination du nombre de clusters à obtenir.

IV.2. La classification dans le contexte de l'aspect mining

La classification a également été appliquée pour l'aspect mining, mais relativement peu. Ainsi, Shepherd et Pollock [Shepherd 05] ont utilisé la classification hiérarchique ascendante pour regrouper les méthodes reliées. L'algorithme proposé commence par mettre chaque méthode dans un cluster séparé, ensuite il fusionne les clusters récursivement, pour lesquels la distance entre les méthodes est inférieure à un certain

seuil. Les auteurs ont utilisé une simple mesure de distance inversement proportionnelle à la longueur des sous-chaînes communes des noms de méthodes. Cette technique a été intégrée dans l'IDE orienté aspect : AMAV (Aspect Miner and Viewer). Toutefois, les auteurs n'ont pas expliqué la manière utilisée pour choisir ou fixer ce seuil.

Zhang et al. [Zhang 08] ont présenté une approche basée sur la classification (CBFA), ayant le fan-in comme critère de base. Le but étant l'identification des candidats aspects. Cette approche a la particularité de regrouper les méthodes relatives à une même préoccupation. Chaque groupe représente un aspect particulier. Le calcul de la métrique fan-in étant effectué pour des groupes de méthodes (clusters). Elle a, par conséquent, l'avantage de capturer des méthodes à faibles fan-in qui auraient pu être omises. Pour cela, l'approche adoptée divise automatiquement l'ensemble de toutes les méthodes du système (toutes classes confondues) en des clusters séparés, où chaque cluster représente un ensemble significatif de méthodes relatives à la même préoccupation. D'après ces auteurs [Zhang 08], ceci permet d'éliminer l'effort requis pour rassembler les méthodes candidates d'une part, et d'améliorer le rappel (Recall) en identifiant des clusters au lieu de méthodes singulières (contrairement au cas de l'analyse fan-in) d'autre part. En effet, les méthodes à faibles fan-in peuvent être identifiées ensemble avec d'autres méthodes reliées ayant des valeurs de fan-in élevées.

He et Bai [He 04] dans leurs travaux se sont basés sur l'hypothèse que si des méthodes apparaissent ensemble dans un nombre différent de modules, cela peut être une bonne indication qu'une préoccupation transverse cachée est présente. La mesure de distance adoptée est une représentation de la dissimilarité des méthodes basée sur les relations d'invocation directe et statiques des méthodes (dans différents contextes).

Dans leurs travaux, Moldovan et al. [Moldovan 06a, Moldovan 06b], la problématique de l'aspect mining est vue comme étant un problème d'identification d'une partition K

d'un système M. Les auteurs [Moldovan 06b] ont alors modélisé le problème d'identification des préoccupations transverses comme étant un problème de classification. Une préoccupation transverse est donc un ensemble de méthodes qui implémentent cette préoccupation. Ils ont présenté une approche d'aspect mining utilisant la classification selon deux techniques : par partition k-means et hiérarchique ascendant.

L'objectif étant l'identification des méthodes qui ont le symptôme de dispersion de code (scattering) : ceci est indiqué par un grand nombre de méthodes appelant une méthode (en se basant sur la méthode fan-in), et aussi par le grand nombre de classes appelantes. Dans leur approche, les objets à classifier sont les méthodes (qui appartiennent aux classes ou appelées à partir de ces classes). Le système est considéré simplement comme un ensemble de méthodes $\{m_1, m_2, \ldots m_n\}$. Pour cela, ils ont considéré deux modèles d'espace vectoriel :

1. Un premier vecteur associé à une méthode M est {FIV, CC} où FIV est la valeur du fan-in et CC le nombre de classes appelantes.
2. Un second vecteur associé à une méthode M est $\{FIV, B_1, B_2, \ldots B_m\}$ où B_i est la valeur de l'attribut correspondant à la classe C_i. La valeur de B_i est égale à 1 si M est appelée depuis une méthode appartenant à C_i, et 0 sinon.

La mesure de similarité utilisée est la distance euclidienne :

$$d(o_i, o_j) = d_E(o_i, o_j) = \sqrt{\sum_{i=1}^{m} \left(o_{it} - o_{jt}\right)^2} \qquad \text{(IV.4)}$$

$$sim(\overrightarrow{o_a}, \overrightarrow{o_b}) = \frac{1}{d(\overrightarrow{o_a}, \overrightarrow{o_b})} \qquad \text{(IV.5)}$$

Dans la version par partition (k-means), le problème revient à chercher une partition optimale qui minimise une fonction objective SSE(k) (Squared Sum Error).

$$SSE(k) = \sum_{k_j \in k} \sum_{o_i \in k_j} d^2(o_i, f_j) \qquad \text{(IV.6)}$$

$$f_j = \left(\frac{\sum_{k=1}^{n_j} o_{k1}^j}{n_j}, \ldots, \frac{\sum_{k=1}^{n_j} o_{km}^j}{n_j} \right) \qquad \text{(IV.7)}$$

Où le cluster $\mathbf{K_j}$ est un ensemble d'objets $\{ \mathbf{O^j_1}, \mathbf{O^j_2}, \ldots, \mathbf{O^j_{nj}} \}$ et $\mathbf{f_j}$ est le centroïde (mean) de $\mathbf{K_j}$.

Quant à la version hiérarchique de leur approche, elle se base sur le résultat de la première version pour trouver le nombre optimal p de clusters. Puis, en se basant sur ce dernier, la version hiérarchique est appliquée dans le but de déterminer exactement p clusters. Ceci est une façon de contourner le problème de déterminations du seuil de troncature de l'arbre hiérarchique.

Selon He [He 06], la classification basée sur les traces d'exécution est appliquée afin de trouver les candidats aspect. Les règles d'association sont ensuite utilisées pour

déterminer la position dans le code source appartenant aux préoccupations transverses (point de coupure) afin de faciliter le « Refactoring ». Les traces d'exécution sont obtenues par instrumentation du système et son exécution sur des scénarios et des entrées spécifiques. Chaque scénario correspond à une séquence d'appels de méthodes.

S'il y a un groupe de code qui a des actions similaires, c.-à-d. une séquence d'appels de méthodes similaire et apparaissant fréquemment dans les traces d'exécution, alors cela peut être un signe de présence de préoccupation transverse. Les séquences d'appels de méthodes similaires constituent de possibles préoccupations transverses. En utilisant la technique de classification sur les scénarios, on considère les scénarios dans les systèmes orientés-objet comme des éléments de données (objets à classifier), les méthodes exécutées par un scénario comme des attributs des objets, et le nombre de fois où chaque méthode est appelée comme les valeurs des attributs.

La matrice des données est formée comme suit :

Pour chaque objet O_i et méthode a_j, si la méthode a_j (j=1,2…,m) est invoquée par l'objet O_i k fois, alors la valeur de la j^{ieme} dimension de l'objet O_i est k. sinon c'est zéro. La matrice des dissimilarités entre les objets stockant la mesure de dissimilarité pour chaque couple d'objets est une matrice n*n. La classification basée sur la matrice des dissemblances d'objets produira un groupe d'ensemble d'objets. Les attributs communs dans chaque ensemble, appelés codes des scénarios, sont les préoccupations transverses candidates.

Une étude comparative a été menée par Czibula [Czibula 07a] sur plusieurs algorithmes de classification. La classification étant utilisée pour identifier une partition du système dans laquelle les méthodes appartenant à une préoccupation transverse devraient être regroupées ensemble. L'étape finale consiste à analyser manuellement les résultats

obtenus. On a considéré le système comme un ensemble de classes $C=\{c_1, c_2,\ldots,c_s\}$, où chaque classe contient une ou plusieurs méthodes. Dans cette approche, les objets à classifier sont les méthodes du système S, $M = \{m_1, m_2, \ldots m_n\}$. L'objectif étant de regrouper les méthodes de telle sorte que les méthodes appartenant à une même préoccupation transverse soient placées dans le même cluster. Afin de regrouper les méthodes dans des clusters, les auteurs ont utilisé quatre algorithmes de classification spécialement définis pour l'aspect mining : kAM introduit dans [Moldovan 06b], HAM dans [Serban 07a], PACO dans [Czibula 09a], et HACO dans [Czibula 09b]. À travers une analyse effectuée sur les quatre algorithmes sur l'étude de cas JHOTDRAW, l'auteur conclut qu'aucun algorithme n'a pu obtenir la partition optimale pour le système. Il conclut, aussi, que la classification hiérarchique semble plus appropriée pour l'aspect mining que les méthodes de partitionnement.

CHAPITRE V

ÉVALUATION DE LA COHÉSION DES CLASSES : UNE NOUVELLE APPROCHE

V.1. L'idée de base

En raison de l'évolution continue, les classes des systèmes orientés objet risquent de devenir volumineuses et impliquer plusieurs fonctionnalités. Une classe investie de plusieurs fonctionnalités (soit dès la conception ou au fur et à mesure de l'évolution) risque d'avoir une faible cohésion et serait plus difficile à comprendre rendant ainsi sa maintenance plus complexe et plus coûteuse. Dans ces conditions, la restructuration devient une mesure nécessaire.

L'objectif de la restructuration consiste essentiellement à accroître la cohésion et à diminuer le couplage. Malgré l'existence de plusieurs approches pour mesurer la cohésion et le couplage, ces dernières ne donnent aucune indication pour restructurer le programme [Lung 04b]. La cohésion réfère à la force interne d'un composant, c.-à-d. la force qui maintient les éléments internes d'un composant ensemble pour effectuer une certaine fonctionnalité. Par conséquent, tous les éléments dans un composant devraient être reliés pour la réalisation d'une certaine responsabilité [Lung 04c].

Le partitionnement joue un rôle crucial dans la conception des systèmes. Il est très relié à la cohésion et au couplage. En fait, la plus importante heuristique du partitionnement est de minimiser le couplage externe et maximiser la cohésion interne [Maier 00]. Le partitionnement consiste à décomposer un système en composants de bas niveau en procédant du haut vers le bas. Elle consiste à regrouper les composants similaires

ensembles pour former des clusters ou des sous-systèmes. Ces derniers sont les partitions qui constituent le système. Il est évident que l'objectif principal du partitionnement et de la classification est le même.

Les méthodes de classification sont concernées par le regroupement des entités en se basant sur leurs inter-relations ou similarités [Wiggerts 97]. Lung [Lung 04a] stipule que « La cohésion est une mesure importante en restructuration. Elle mesure à quel degré les éléments d'un composant sont reliés entre eux. Or, le but de la classification est de regrouper des éléments similaires ou reliés ensemble. Il est alors possible d'utiliser la classification pour mesurer la force des relations (liaison) entre les éléments dans un composant ».

Le but de la classification est de différencier les groupes au sein d'un ensemble d'objets donné, selon un certains nombre de caractéristiques ou d'attributs appropriés des objets analysés. Chacun de ces sous-ensembles (clusters) consiste en des objets qui sont similaires entre eux et dissemblables par rapport aux objets des autres groupes [Moldovan 06b].

Par ailleurs, Lung et Zaman [Lung 04b] précisent que leur approche de classification peut évaluer la cohésion d'une fonction et peut donner des indications sur comment décomposer une large fonction en de multiples fonctions à forte cohésion. Ils affirment même que la technique de classification peut être utilisée pour mesurer la cohésion d'un programme et assister les activités de restructuration des programmes (au niveau des fonctions). La classification peut être facilement appliquée à des niveaux variés d'abstraction. Elle peut fournir une valeur ajoutée en facilitant la restructuration des systèmes [Lung 04c].

En accord avec Lung et Zaman, nous croyons qu'il est possible d'utiliser la classification non seulement pour mesurer la cohésion des fonctions dans les systèmes procéduraux, mais aussi au niveau des classes qui sont les composants logiciels de base dans les systèmes orienté-objet. C'est effectivement l'objectif principal de notre travail, car le principe et l'objectif de la classification coïncident avec ceux de la cohésion.

V.2. Présentation de la nouvelle approche de calcul de la cohésion

Notre objectif étant d'utiliser la classification afin d'améliorer l'évaluation de la cohésion des classes dans les systèmes orientés objet, de sorte à mieux refléter la qualité conceptuelle des classes, à déterminer les différents groupes cohésifs dans une classe et déceler la disparité dans le code d'une classe. Cela nous permettra de déceler des problèmes de conception liée à la mauvaise affectation de responsabilités à une classe, et éventuellement déceler la présence de code pouvant correspondre à des préoccupations transverses et, par conséquent, mieux orienter les actions de restructuration du système analysé.

L'objectif consiste à regrouper les membres similaires (ou reliés) d'une classe en utilisant le principe de la classification. À cet effet, le choix de l'ensemble des propriétés caractérisant chaque objet (membre) à classifier est déterminant. Il faut donc décider :

a) Quels sont les types d'objets qui doivent être mesurés.
b) Quel type de propriétés doit être considéré pour un concept de similarité.

Dans ce contexte, les objets à classifier sont les membres d'une classe : méthodes et attributs (déclarés explicitement dans la classe). Quant au choix de l'ensemble des propriétés, on s'est basé sur les critères de cohésion (pour une première exploration).

Sachant qu'une méthode peut utiliser des variables d'instance ou appeler d'autres méthodes, et qu'une variable d'instance peut être utilisée par des méthodes, l'ensemble des propriétés sera donc formé des variables d'instance et des méthodes de la classe.

Il s'agit là de la principale caractéristique qui distingue notre approche par rapport aux travaux de Czibula et al. [Czibula 07a, Czibula 07b, Czibula 07c, Czibula 07d]. En effet, dans ces travaux, les éléments à classifier étant l'ensemble formé par toutes les classes et les membres (méthodes et attributs) de toutes les classes du système (en ignorant complètement la répartition en classes établie par le concepteur du système). Aussi, leur objectif est de trouver une meilleure restructuration du système (niveau global), en cherchant une meilleure classification des méthodes et des attributs de tout le système en clusters qui vont représenter les nouvelles classes du système analysé.

Dans le cadre de notre approche, nous nous concentrons sur l'analyse de chaque classe indépendamment des autres classes. Le but est d'évaluer son unité structurelle et fonctionnelle et de déceler la disparité dans ses responsabilités.

V.2.1. Définition du modèle

L'élaboration de notre modèle comporte essentiellement les trois étapes suivantes :

1. L'élaboration de la matrice entité propriétés,
2. La classification hiérarchique,
3. La détermination du nombre optimal de clusters.

1) Élaboration de la matrice entité propriétés

Soit une classe C du système à analyser, pour une entité donnée e de la classe C, p(e) représente un ensemble de propriétés appropriées caractérisant l'entité e définies comme suit :

- Si e est un attribut, alors p(e) correspond à : l'attribut lui-même et toutes les méthodes de la classe C qui utilisent cet attribut.
- Si e est une méthode, alors p(e) correspond à: la méthode elle-même, tous les attributs de la classe C référencés par cette méthode et toutes les méthodes de la classe C utilisées par cette méthode.

Le degré de similarité entre deux entités d'une classe C, e_i et e_j est donné par l'équation suivante :

$$s(e_i, e_j) = \frac{| p(e_i) \cap p(e_j) |}{| p(e_i) \cup p(e_j) |} \qquad \text{(V.1)}$$

S prend ses valeurs dans l'intervalle [0..1].

Ainsi, nous introduisons la matrice entité-propriétés. Cette matrice est une matrice binaire (K+L) * (K+L), où K est le nombre de méthodes et L est le nombre d'attributs de la classe considérée. Pour construire cette matrice, les noms des méthodes et des attributs sont extraits depuis le code source de la classe en question. Cette matrice modélise les différentes interactions : méthode-attribut, méthode-méthode et attribut-attribut.

Il y a une interaction attribut-méthode (ou méthode-attribut) si l'attribut figure dans le corps de cette méthode. Il y a une interaction méthode-méthode si une méthode appelle l'autre. Enfin, il y a interaction avec l'attribut lui-même.

Toute interaction est représentée par la valeur 1 dans la case correspondante de la matrice, sinon la case est mise à 0. La matrice résultante est fournie en entrée (comme input) pour l'algorithme de classification hiérarchique, afin d'obtenir plusieurs partitions emboîtées des membres de la classe, sous forme d'un arbre hiérarchique (l'algorithme est déjà implémenté dans plusieurs outils statistiques en l'occurrence SPSS, Xlstat, ..). Dans notre cas, nous avons choisit celui intégré à l'outil XLstat de Addinsoft.

2) La classification hiérarchique

La matrice entité propriétés est soumise à l'algorithme de la classification hiérarchique décrit par les étapes suivantes:

1- *Choisir un type de mesure de similarité adapté au sujet étudié et à la nature des données : l'indice de Jaccard dans notre cas.*

2- *Calculer la matrice des dissimilarités en calculant la dissimilarité (par paire) entre les N entités.*

Répéter les étapes (3) et (4) jusqu'à ce que toutes les entités soient regroupées

3- *Regrouper les deux entités dont le regroupement minimise un critère d'agrégation donné, créant ainsi un cluster comprenant ces deux entités.*

4- *Mettre à jour la matrice des dissimilarités en calculant la dissimilarité entre ce nouveau cluster et les autres entités non encore fusionnées restantes en utilisant le critère d'agrégation.*

Itérer.

3) La détermination du nombre optimal de clusters.

Enfin, il s'agira de déterminer le nombre optimal de clusters (ou partitions) ou le choix optimal du niveau de troncature approprié de l'arbre hiérarchique en choisissant une méthode parmi celles citées précédemment dans la section III.5.

Dans notre étude, la troncature automatique intégrée à XLSTAT, de l'arbre hiérarchique à un niveau approprié détermine le nombre optimal de clusters de la classe analysée du système.

V.2.2. Définition de la nouvelle métrique de cohésion

L'approche que nous proposons permet d'obtenir deux résultats : d'une part, le nombre optimal de clusters qui nous renseigne sur **le nombre de composantes connexes** au sein de la classe, et d'autre part, le graphe modélisant les dépendances qui sera utilisé pour déterminer le nombre de méthodes connectées afin de calculer la **cohésion** par rapport au nombre total de connections possibles du graphe. Sachant que chaque cluster regroupe uniquement des entités reliées, les paires de méthodes, à l'intérieur d'un cluster, sont connectées. On pourra, alors, déduire le graphe de connexion G des méthodes de la classe analysée selon la partition résultante du processus de la classification. Ce dernier (c.-à-d. le graphe résultant de la classification) servira de base pour définir une première métrique relative à la cohésion qu'on désignera par $\mathbf{COH_{CL}}$

La définition de la cohésion d'une classe sera basée sur le degré de liaison entre ses méthodes. Telles que décrites initialement dans la section II.2, ces métriques sont définies en termes de méthodes reliées. À ce niveau, nous ignorons les clusters qui contiennent uniquement des variables d'instance, puisque ce sont les connectivités des méthodes qui nous intéressent pour l'évaluation de la cohésion.

Soit N_{tot} le nombre total de paires de méthodes d'une classe C, c.-à-d. le nombre maximal de connexions entre ses méthodes. Ainsi, pour une classe possédant N méthodes :

$$\mathbf{N_{tot} = N*(N-1) / 2} \qquad (V.2)$$

Considérons un graphe non dirigé G résultant de la classification où chaque nœud représente une méthode de la classe. Il y a un arc entre deux méthodes M_i et M_j si elles appartiennent au même cluster (reliées). Ce graphe schématise les connectivités entre les méthodes de la classe analysée. Soit N_C le nombre de connexions entre les méthodes qui est obtenu en calculant le nombre d'arcs dans le graphe G.

COH_{CL} est définie par le nombre relatif de méthodes connectées au sein de la classe et correspond à:

$$\mathbf{COH_{CL} = Nc / N_{tot}} \qquad (V.3)$$

$\mathbf{COH_{CL}} \in [0,1]$.

Du même coup, une seconde métrique aussi importante que la première est aussi obtenue. Il s'agit du Nombre de Composantes Connexes **NCC** dans la classe qui n'est rien d'autre que le nombre de clusters (par omission des clusters contenant uniquement des variables d'instances).

Afin de mieux introduire notre approche, nous allons l'illustrer à travers un exemple abstrait.

V.3. Exemple de calcul de la cohésion selon la nouvelle approche

Soit C une classe contenant les attributs et les méthodes suivants :

A_C= {i_k ; 1=<k<=5} : l'ensemble des attributs (i_1, i_2, …, i_5).
M_C= {m_k ; 1=<k<=7} : l'ensemble des méthodes (m_1, m_2, …, m_7.)
$C=A_C \cup M_C$

Soit U_j : l'ensemble des attributs et des méthodes utilisées par une méthode j.

Supposons que l'utilisation des attributs et des méthodes est faite selon la distribution suivante :

U1= { i1, i2, i3, m3 }, U2= { i1, i2, m3 } ,
U3= { i1, i2 } , U4= { i1 },
U5= { i1, i4, i5 }, U6= { i4, i5 } et U7= { i4 }

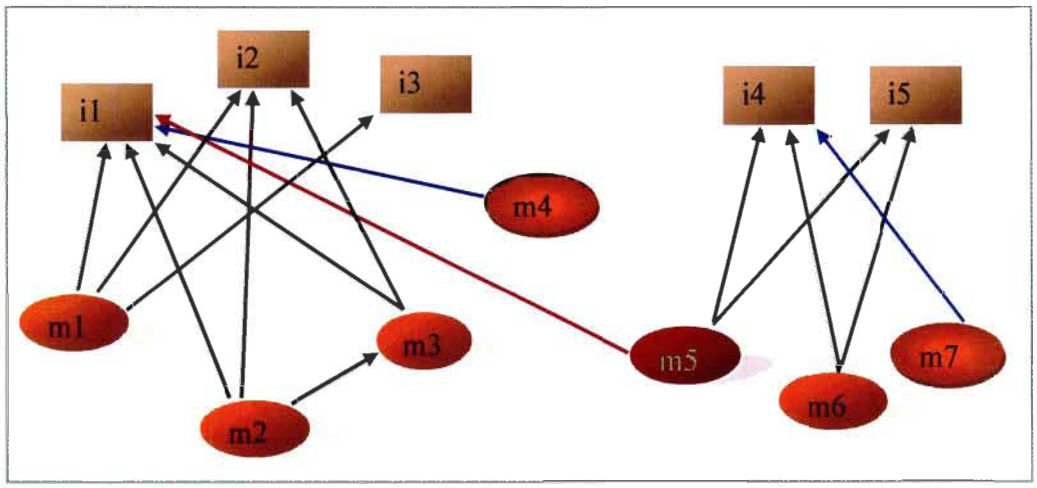

Figure V.1 : Schéma d'utilisation des attributs et des méthodes de la classe C.

La figure V.1 présente les différentes connexions qui existent entre les éléments de la classe C selon les données des ensembles U_i : 1<= i<= 7.

1- Définition de la matrice Entité-propriétés

Cette matrice (Tableau V.1) est déterminée selon la définition du modèle présenté précédemment. Elle a donc pour lignes les entités qu'on veut classifier (tous les attributs et les méthodes membres de la classe analysée), et pour colonnes les propriétés qui caractérisent chaque entité à classifier (les variables d'instances et les méthodes utilisées par les méthodes de cette classe). Par exemple, puisque la méthode m_4 utilise l'attribut i_1 (figure V.1), nous avons mis 1 dans la case (m4 : i1) de la matrice entité-propriétés (Tableau V.1). Le même principe est appliqué pour les autres paires dans la matrice.

	propriétés											
Entités	i_1	i_2	i_3	i_4	i_5	***m1***	***m2***	***m3***	***m4***	***m5***	***m6***	***m7***
m_1	1	1	1	0	0	0	0	1	0	0	0	0
m_2	1	1	0	0	0	0	0	1	0	0	0	0
m_3	1	1	0	0	0	0	0	0	0	0	0	0
m_4	1	0		0	0	0	0	0	0	0	0	0
m_5	1	0	0	1	1	0	0	0	0	0	0	0
m_6	0	0	0	1	1	0	0	0	0	0	0	0
m_7	0	0	0	1	0	0	0	0	0	0	0	0
i_1	1	0	0	0	0	1	1	1	1	1	0	0
i_2	0	1	0	0	0	1	1	1	0	0	0	0
i_3	0	0	1	0	0	1	0	0	0	0	0	0
i_4	0	0	0	1	0	0	0	0	0	1	1	1
i_5	0	0	0	0	1	0	0	0	0	1	1	0

Tableau V.1 : Matrice Entités-propriétés de la classe C.

2- Choix d'une mesure de similarité appropriée

Avant d'appliquer un algorithme de classification quelconque, il est nécessaire de choisir ou de définir une mesure de similarité. Plusieurs études ont montré que le coefficient de Jaccard est plus approprié pour les données binaires que les autres coefficients. Anquetil et al. [Lethbridge 99] ont déterminé, lors d'une étude visant à décomposer des systèmes linux, Mosaic, gcc et autres, que le coefficient de Jaccard tend à donner de meilleurs résultats que le coefficient simple.

Nous avons donc retenu le **coefficient de Jaccard** pour illustrer cet exemple et également pour réaliser toutes nos expérimentations.

3- Classification hiérarchique

3.1) Calcul de la matrice des proximités

On commence par calculer la dissimilarité entre toute paire d'objets. Ici, les objets sont les méthodes m_i et les attributs i_k . Le nombre d'objets **N** est donc égal à 12 dans l'exemple considéré. Dans le cas des données binaires, La matrice des dissimilarités se déduit selon la formule suivante :

$$\text{dissimilarité} = 1 - \text{similarité} \qquad \text{(V.4)}$$

Par exemple, pour calculer la valeur de proximité entre les méthodes m1 et m2 selon l'indice de Jaccard (voir Tableau III.1 de la section III.3.3), La similarité est donnée par la formule (V.5).

$$S(m1, m2) = a / (a + b + c) \qquad \text{(V.5)}$$

Où

- a : représente le nombre de correspondances (1,1) des méthodes (m1, m2) respectivement dans le tableau V.1.
- b : représente le nombre de correspondances (1,0) des méthodes (m1, m2) respectivement dans le tableau V.1.
- c : représente le nombre de correspondances (0,1) des méthodes (m1, m2) respectivement dans le tableau V.1.

Ce qui donne : $a = 3$, $b = 1$ et $c = 0$ d'où

$S(m1, m2) = 3/(3+1+0) = ¾ = 0.750$

Par conséquent, on obtient la dissimilarité comme suit:

$d(m1, m2) = 1 - 0.750 = 0.250.$

Enfin, la matrice des proximités (tableau V.2) qui en résulte donne le degré de dissimilarité entre toutes les paires d'entités.

Entités	*m1*	*m2*	*m3*	*m4*	*m5*	*m6*	*m7*	*i1*	*i2*	*i3*	*i4*	*i5*
m1	0,000	**0,250**	0,500	0,750	0,833	1,000	1,000	0,750	0,667	0,800	1,000	1,000
m2	**0,250**	0,000	0,333	0,667	0,800	1,000	1,000	0,714	0,600	1,000	1,000	1,000
m3	0,500	0,333	0,000	0,500	0,750	1,000	1,000	0,857	0,800	1,000	1,000	1,000
m4	0,750	0,667	0,500	0,000	0,667	1,000	1,000	0,833	1,000	1,000	1,000	1,000
m5	0,833	0,800	0,750	0,667	0,000	0,333	0,667	0,875	1,000	1,000	0,833	0,800
m6	1,000	1,000	1,000	1,000	0,333	0,000	0,500	1,000	1,000	1,000	0,800	0,750
m7	1,000	1,000	1,000	1,000	0,667	0,500	0,000	1,000	1,000	1,000	0,750	1,000
i1	0,750	0,714	0,857	0,833	0,875	1,000	1,000	0,000	0,571	0,857	0,889	0,875
i2	0,667	0,600	0,800	1,000	1,000	1,000	1,000	0,571	0,000	0,800	1,000	1,000
i3	0,800	1,000	1,000	1,000	1,000	1,000	1,000	0,857	0,800	0,000	1,000	1,000
i4	1,000	1,000	1,000	1,000	0,833	0,800	0,750	0,889	1,000	1,000	0,000	0,600
i5	1,000	1,000	1,000	1,000	0,800	0,750	1,000	0,875	1,000	1,000	0,600	0,000

Tableau V.2 : Matrice des proximités de la classe C.

3.2) Les deux éléments les plus similaires (proches) sont m1 et m2 à distance 0.250 (valeur en surbrillance dans le tableau V.2). Cela correspond à la valeur minimale dans la matrice. On agrège ces deux méthodes en créant ainsi un nœud (ou un groupe) **c1= (m1, m2).**

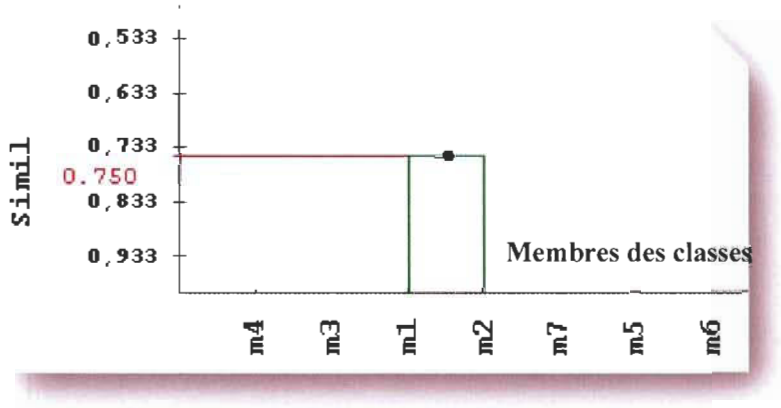

Figure V.2 : Fusion de deux méthodes pour créer un nœud.

3.3) La matrice des dissimilarités est ensuite mise à jour comme suit :

On calcule la dissimilarité entre ce nouveau groupe **c1** et les N-2 autres méthodes restantes en utilisant le critère d'agrégation choisi précédemment (**Lien simple**). Ainsi, la dissimilarité entre A et B est la dissimilarité entre l'objet A et l'objet B les plus ressemblants (proches) c.-à-d. ayant la plus petite dissimilarité. Donc, on prend **le minimum** (voir tableau V.3).

Par exemple :

d(m3,m1)=0.500 et d(m3,m2)=0.333
et comme c1=(m1,m2) alors;

d(m3,c1) =min(d(m3,m2), d(m3,m1))= min(0.333, 0.500) = 0.333

De même,

d(m4,m1)=0.750 et d(m4,m2)=0.667

d(m4,c1) =min(d(m4,m2), d(m4,m1))= min(0.667,0.750) = 0.667

Et ainsi de suite…

Entités	*c1*	*m3*	*m4*	*m5*	*m6*	*m7*	*i1*	*i2*	....
c1	0	**0,333**	0,667	0,800	1,000	1,000	…	…	…
m3	0,333	0	0,500	0,750	1,000	1,000	…	…	…
m4	0,667	0,500	0	0,667	1,000	1,000	…	…	…
m5	0,800	0,750	0,667	0	0,333	0,667	…	…	…
m6	1,000	1,000	1,000	0,333	0	0,500	…	…	…
m7	1,000	1,000	1,000	0,667	0,500	0	…	…	…
i1	…	…	…	…	…	…	…	…	…
i2	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	..	…	…	…

Tableau V.3 : Mise à jour de la Matrice des proximités après fusion.

3.4) L'algorithme est récursif. La fusion des clusters se fait récursivement. À chaque itération on fusionne les deux clusters les plus similaires (proches) et ainsi de suite jusqu'à aboutir à un seul cluster. À chaque fusion de deux clusters on crée un nœud.

Le tableau V.4 de la page qui suit, donne les statistiques des nœuds pour les agrégations successives.

Nœud	*Niveau*	*Objets*	*Fils gauche*	*Fils droit*
23	0,963	12	22	21
22	0,91	7	20	10
21	0,822	5	17	18
20	0,778	6	19	16
19	0,639	4	15	4
18	0,6	2	11	12
17	0,583	3	14	7
16	0,571	2	8	9
15	0,417	3	13	3
14	0,333	2	5	6
13	0,25	2	1	2

Tableau V.4 : Statistiques des nœuds (la classe *C*).

Le tableau V.4 affiche les informations concernant les nœuds successifs du dendrogramme. Le premier nœud a pour indice le nombre d'entités à classifier augmenté de 1 (comme ici 13=12+1). Ainsi, il est aisé de repérer à quel moment une entité ou un groupe d'entités est regroupé avec une autre entité ou groupe.

Il faut noter que XLSTAT crée un index interne pour les entités à classifier. Les entités sont indexées en ordre croissant par rapport à leur position dans la matrice entité-propriétés. L'index représente, en fait, le numéro de ligne en commençant par 1. Par exemple, la méthode m1 aura comme index 1, la méthode m2 l'index 2 et ainsi de suite. La variable d'instance i5 aura l'index 12 (ligne 12 de la matrice (Tableau V.2)).

Pour lire le tableau de statistiques des nœuds (Tableau V.4), il faut commencer par le bas du tableau vers le haut. Ainsi, la dernière ligne de ce tableau, par exemple, indique qu'une fusion (ou agrégation) des deux entités 1 et 2 (respectivement (m1) comme fils gauche et (m2) comme fils droit) est effectuée pour créer un nœud qui sera indexé 13

(pour continuer la numérotation précédente). Ce nœud est situé à un niveau de similarité de 0.25. La première ligne du tableau représente la dernière fusion qui aboutit à un seul cluster contenant toutes les entités. Dans ce cas, le nœud créé se situe à un niveau de similarité égal à 0.963.

L'arbre hiérarchique résultant est présentée à la figure V.3. Il correspond, en fait, à la représentation graphique (plus conviviale) des nœuds résultant des fusions successives présentées précédemment dans le tableau des statistiques des nœuds. L'outil utilisé pour réaliser la classification hiérarchique lors des expérimentations est XLSTAT. Ce dernier peut déterminer automatiquement le nombre de clusters. Le critère utilisé pour la troncature est basé sur l'entropie.

Pour un niveau de troncature automatique selon XLSTAT, du point de vue classification, on obtient 3 clusters distincts :

{m1, m2, m3, m4, i1, i2} , {i3} et {m5, m6, m7, i4, i5}.

Cependant, du point de vue cohésion de la classe, puisque ce qui nous intéresse ce sont les connectivités entre méthodes (par omission des variables d'instances), on obtient en fait deux clusters :

{m1, m2, m3, m4} et {m5, m6, m7}.

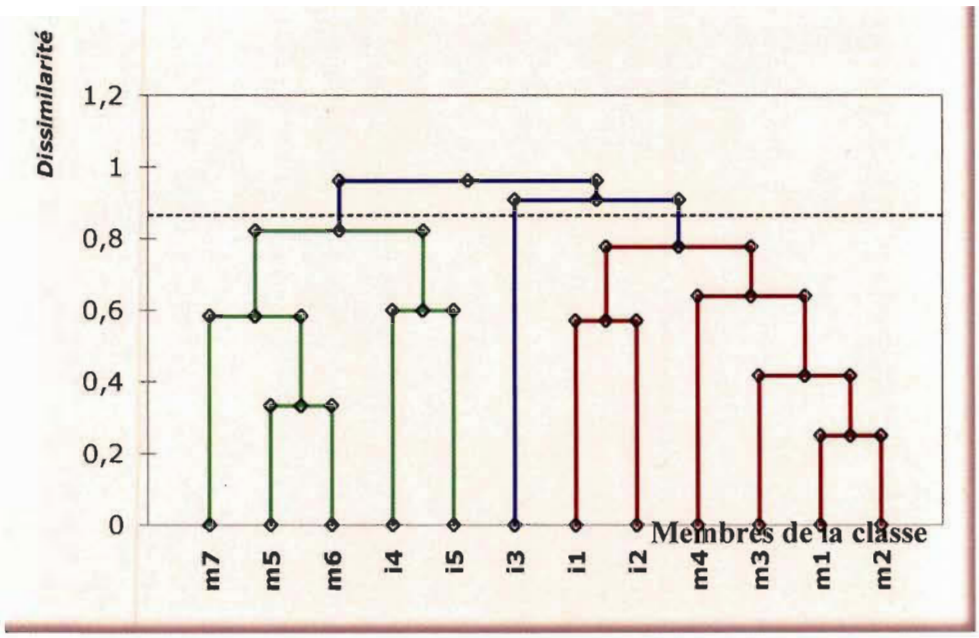

Figure V.3 : L'arbre hiérarchique résultant pour la classe C.

Toutes les méthodes d'un même cluster sont reliées. Par conséquent, on peut déduire le graphe G de connexion issu de la classification (figure V.4).

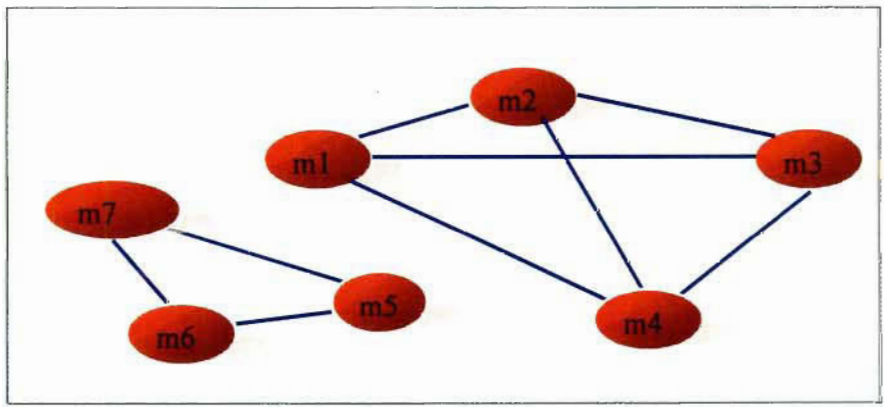

Figure V.4 : Graphe G de connexions des méthodes pour la classe C.

Donc, d'après le graphe de la figure V.4, la cohésion selon la nouvelle approche est : COH_{CL}= **9/21=0.42.** À partir du même graphe on pourrait aussi déduire le nombre de composantes connexes déterminées par la classification qui est, en fait, le nombre optimal de clusters obtenus. Dans cet exemple, le nombre de composantes connexes est NCC= 2.

En résumé, pour mesurer la cohésion d'une classe, la nouvelle approche utilise la classification en considérant tous les critères de cohésion. Tous les critères de cohésion reflétant les interactions possibles entre les membres d'une classe ont été impliqués dans la définition des données de base du processus de classification, à savoir : les interactions méthode-attribut, méthode-méthode. Selon le niveau de troncature déterminé par l'outil XLSTAT, on obtient le nombre optimal de clusters existants dans la classe. Ce nombre nous renseigne sur le nombre de composantes connexes dans la classe. À partir de la répartition résultante du processus de classification, on déduit un graphe de connectivité des méthodes pour la classe en question. À partir de ce graphe, nous pouvons calculer la cohésion comme étant le rapport du nombre de paires de méthodes connectées sur le nombre total de paires de méthodes.

V.4. Interprétation de la nouvelle métrique

Les différentes composantes connexes au sein d'une classe peuvent refléter, dans certains cas, la disparité dans l'affectation des rôles à une classe. **NCC** peut être utilisé alors comme indicateur de disparité des fonctionnalités offertes par une classe ou dans le pire des cas, signaler un défaut de conception.

En effet, une valeur de NCC égale à 1 indique que toutes les méthodes de la classe sont reliées et forment un seul groupe connexe, travaillant conjointement pour représenter un

comportement unique et bien défini, donc une seule responsabilité. Alors que toute valeur supérieure à 1 de NCC, indique que la classe a plusieurs groupes disparates et qu'elle supporte probablement plusieurs responsabilités. Il serait préférable, dans ce cas, d'examiner la possibilité de restructurer la classe en séparant les groupes de méthodes connexes en autant de préoccupations distinctes (classes) du moment que les méthodes appartiennent à des ensembles disjoints.

Quant à la métrique $\mathbf{COH_{CL}}$, définie comme étant le pourcentage du nombre de paires de méthodes connectées dans une classe par rapport au nombre maximal de connexions possibles entre les méthodes de la classe, sa valeur exprime le degré de liaison entre les membres d'une classe. Une valeur faible indique que ses membres sont faiblement reliés entre eux, bien qu'ils puissent constituer un seul groupe représentant un comportement bien défini. Toutefois, elle peut aussi implicitement indiquer l'existence de plusieurs groupes connexes. D'ailleurs, une faible valeur de $\mathbf{COH_{CL}}$ peut être interprétée de diverses façons et révéler diverses situations :

1- Les membres de la classe constituent un seul groupe de membres connectés, mais faiblement reliées.
2- Les rôles assignés à la classe sont disparates (non reliés).
3- Potentiellement, les deux situations précédentes.

Il faut noter que nous pouvons avoir, par exemple, deux classes avec des valeurs de cohésion semblables sauf que dans le premier cas, les membres sont faiblement reliés, mais constituent un seul groupe de membres connectés, et dans le second cas les rôles assignés à la classe sont disparates. Alors, il nous faut un autre moyen pour distinguer entres les différentes situations précédentes. C'est justement le rôle de la métrique NCC qui devrait être utilisée conjointement avec la métrique COH_{CL}.

En effet, si on considère l'exemple précédent (figure V.4) de la section V.3, la valeur de la cohésion est assez faible COH_{CL}=0.42. Par ailleurs, sachant que la valeur de la métrique NCC est égale à 2, cela explique clairement que la classe donnée en exemple modélise deux fonctionnalités non reliées (du moins à travers les connexions modélisées). Sans la métrique NCC, on n'aurait pas pu expliquer le résultat obtenu de la cohésion (révéler explicitement l'existence de deux groupes non reliés).

Il est également possible, d'avoir une classe ayant une cohésion assez forte (selon les métriques de cohésion existantes y compris COH_{CL}), bien que la classe comporte deux groupes connexes distincts. En effet, imaginons par exemple le cas d'une classe C2 de la figure V.5 comportant en tout 11 méthodes telles que : 2 de ces méthodes (m1, m2) forment un seul groupe et les neuf restantes (m3, m4....,m11) constituent un autre groupe complet, sauf qu'il n'y a aucun lien entre les deux groupes. Alors, selon une approche d'évaluation de cohésion telle que $DC_{IE,}$ par exemple, la cohésion de cette classe serait égale à 0.67 (=37/55). Même du point de vue de notre approche, la cohésion de cette classe donne la même valeur pour la métrique COH_{CL}, qui est quand même une cohésion assez bonne, pourtant on voit bien qu'elle modélise deux groupes complètement disjoints.

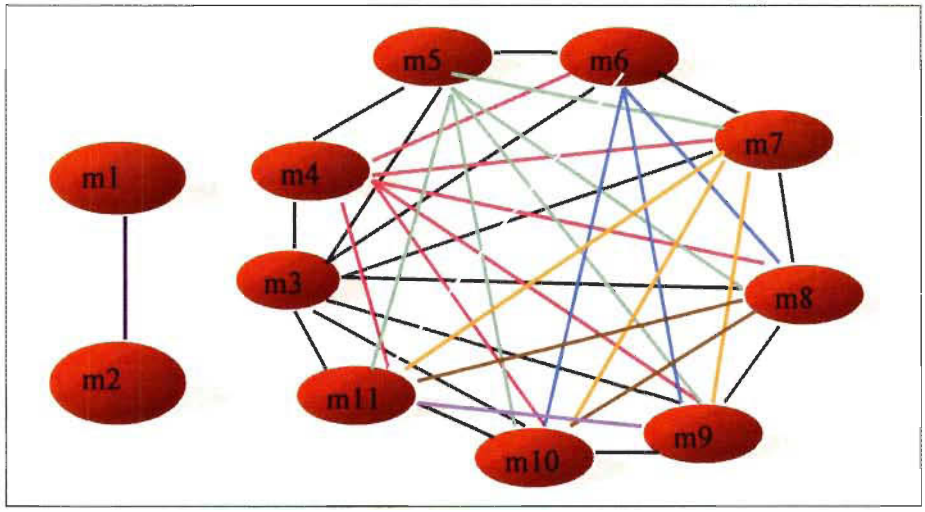

Figure V.5 : Graphe de connexions des méthodes pour une classe C2.

(N.B. Dans ce cas précis, il s'agit peut-être simplement d'un défaut de conception.)

Sans la métrique **NCC**, il est impossible d'identifier ces problèmes de conception sans l'inspection manuelle du code source. La métrique NCC aide à diagnostiquer explicitement cette problématique. Alors, il est plus judicieux d'utiliser conjointement les deux métriques afin de mieux distinguer entre plusieurs situations relatives à des défauts de conception dans les classes et mieux interpréter ces défauts.

CHAPITRE VI

ÉVALUATION EMPIRIQUE DE LA NOUVELLE APPROCHE

VI.1. Introduction

Pour évaluer notre approche, nous avons mené une étude expérimentale au cours de laquelle nous avons considéré plusieurs exemples de classes (en java). Ces exemples ont été minutieusement sélectionnés. La plupart ont été largement discutés dans la littérature spécialisée dans les domaines de la maintenance, de la restructuration et de l'aspect mining. Pour réaliser cette étude, nous avons utilisé l'outil XlSTAT de Addinsoft. Il intègre plusieurs utilitaires de statistiques et d'analyse de données, dont l'utilitaire de classification hiérarchique et l'option de troncature automatique de l'arbre hiérarchique sur laquelle est fondée notre approche. Pour chaque exemple, la démarche sera la suivante :

1. Présentation de l'exemple et discussion (analyse) des « défauts » de conception qu'il présente;
2. Évaluation de la cohésion de la classe selon l'ancienne approche, en utilisant la métrique DC_{IE};
3. Évaluation de la cohésion de la classe selon la nouvelle approche (basée sur la classification). Cette phase inclut les étapes suivantes :

- Déterminer la matrice entité-propriétés selon les spécifications du modèle de la nouvelle approche.

- Fournir cette matrice à l'utilitaire de classification hiérarchique en choisissant comme mesure de similarité l'indice de Jaccard.
- Obtenir la matrice des proximités et les statistiques des nœuds.
- Obtenir l'arbre hiérarchique associé à la classe analysée.
- Obtenir, selon la troncature automatique de l'arbre, un ensemble de clusters.
- Filtrage des clusters pour obtenir seulement les clusters de méthodes.
- Obtenir la valeur de la métrique NCC.
- Obtenir le nouveau graphe de connectivités des méthodes selon la nouvelle approche.
- Calculer la métrique de cohésion COH_{CL}.

À la fin de chaque exemple, nous donnons une brève discussion. Cette discussion sera basée sur les défauts (ou faiblesses) que présente l'exemple, les valeurs obtenues par notre approche, et les liens que nous pouvons faire entre ces valeurs et la manière de les utiliser pour capturer les défauts et proposer éventuellement une restructuration des classes.

On mettra également en évidence la différence entre les deux approches (ancienne et nouvelle) sur le plan des capacités à détecter (dans le sens refléter) ces problématiques.

Dans ce qui suit, nous décrirons chacun de ces exemples. Le premier exemple sera décrit en détail. Pour les trois autres on présentera seulement les détails pertinents sans toutefois trop s'étaler.

VI.2. Exemple 1 (Design Pattern Observer)

VI.2.1. Présentation de l'exemple

Il s'agit d'une implémentation du patron Observateur [Hannemann 02]. Les patrons de conception visent à introduire au sein du développement de logiciels les meilleures pratiques de conception. Le patron Observateur consiste à définir une dépendance « un à plusieurs » entre les objets : un sujet vis-à-vis de plusieurs observateurs. De cette façon, lorsque l'objet sujet change d'état, tous les objets observateurs seront automatiquement avisés et mis à jour en conséquence.

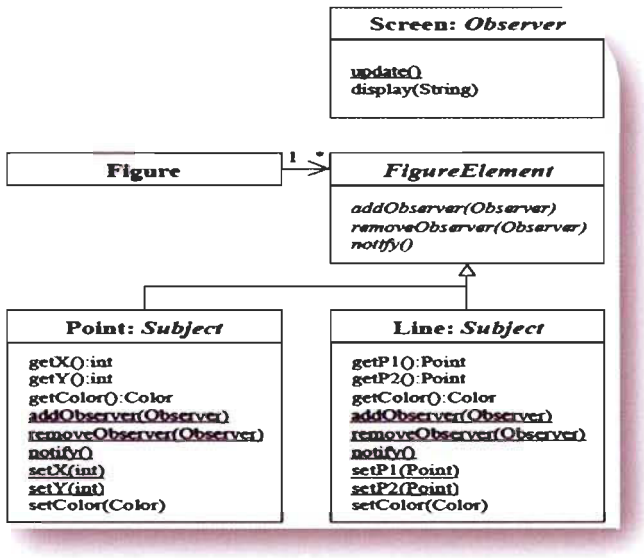

Figure VI.1.1 : Diagramme UML de l'exemple du pattern Observer [Hannemann 02].

Cet exemple tel qu'illustré par la figure VI.1.1 contient deux classes, *Point* et *Line*, qui partagent une même interface *FigureElement*. La classe *Line* est composée de deux objets de type *Point*. Les méthodes préfixées par set permettent de modifier l'état de l'objet instancié pour chaque classe. Si on considère ces classes comme faisant partie

d'un logiciel de dessin, alors la création d'un point ou d'une ligne dans l'éditeur aura pour effet de créer et d'associer ces deux nouveaux objets à une représentation graphique à l'écran. Si un appel à l'une des méthodes préfixées par set sur un objet de type *FigureElement* est déclenché, l'objet observateur doit être averti afin de pouvoir répercuter les modifications au niveau de la représentation graphique.

Une solution simple est de faire une mise à jour des objets observateurs directement depuis chacune des méthodes préfixées par set. Cependant, cela entraine des problèmes en termes de réutilisation ou de maintenance. Ce genre de problème a été étudié par la communauté du génie logiciel et a donné naissance aux patrons de conception. Selon l'approche objet, cela implique généralement que le sujet (l'objet observé) doit définir un champ (liste) pour fournir la mécanique d'inscription et de désinscription des observateurs intéressés (méthodes *addObserver()* et *removeObserver()*) ainsi qu'une méthode de notification *notify()*.

L'interface Observer doit être implémentée par les classes des observateurs. Cette interface comprend une méthode, update, qui est appelée pour notifier à l'observateur un changement au niveau du sujet d'observation. Voila une implémentation OO possible de cet exemple (voir figure VI.1.2).

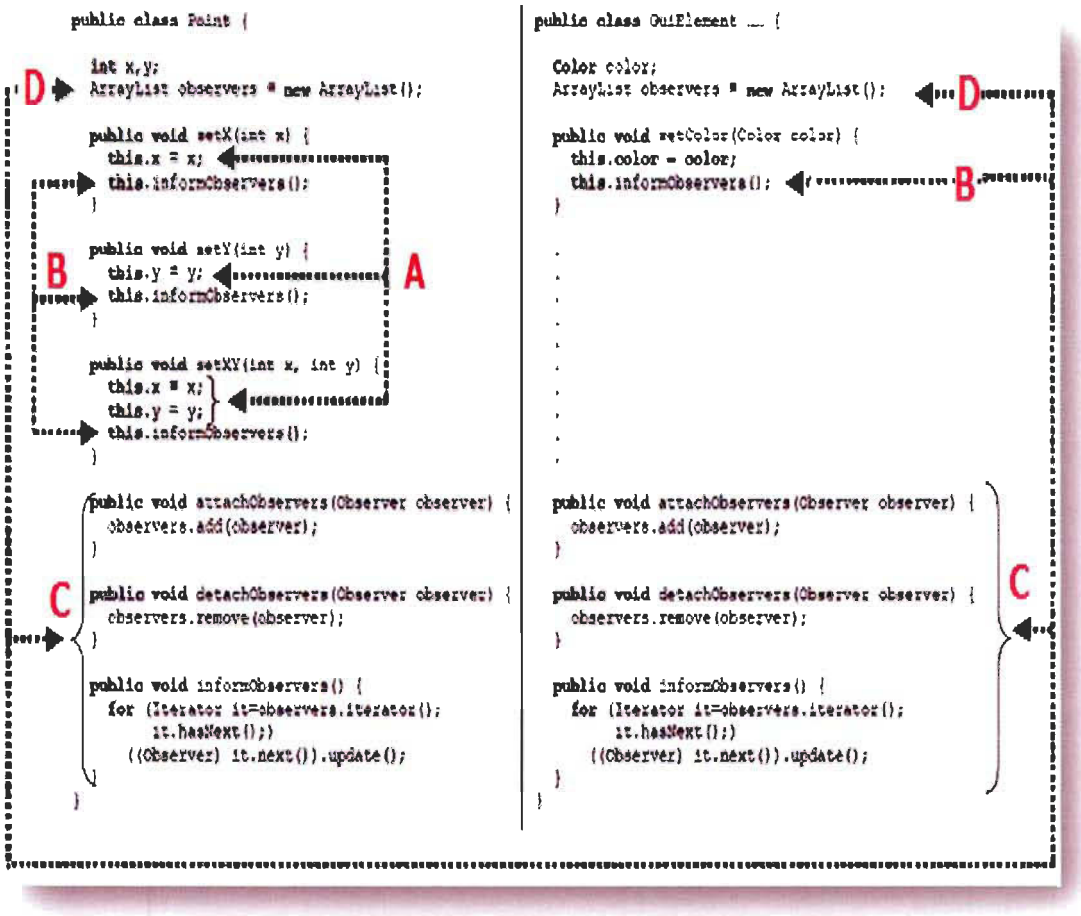

Figure VI.1.2 : Implémentation du patron Observateur- code dispersé et enchevêtré.

A partir de cet exemple, on constate clairement, les limites de l'approche Objet dues à la duplication et la dispersion du code, d'une part, et à l'enchevêtrement du code d'autre part qui étaient inévitables.

En effet, la classe sujet : *Point* doit intégrer tout le fragment de code relatif à la tenue à jour et à la notification des objets observateurs dans son implémentation (bouts de code étiquetés **C** et **D** sur la figure VI.1.2).

Aussi, on note la grande dispersion du code résultant de l'appel à la méthode *informObservers()* pour signaler le changement subit sur un attribut à l'objet observateur (bout de code étiqueté **B**) qu'on trouve aussi bien dans les différentes méthodes préfixés par set à l'intérieur de la même classe (*Point*) qu'à l'extérieur dans l'autre classe (*GuiElement*). D'autre part, on note l'enchevêtrement du code (tangling) dans la même classe (*Point)*, entre sa fonctionnalité principale (bout de code étiqueté **A)** mixé avec le code relatif au mécanisme de notification de l'observateur que nous avons décrit ci-haut (code étiqueté **B, C, D).**

Cela signifie que la classe *Point* a été dotée de plus d'une responsabilité contrairement à ce qu'il devrait être en principe. Le code de base de la classe *Point* aurait du ressembler au suivant (figure VI.1.3) :

```
public class Point {
  int x, y;

  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }

  public void setXY(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Figure VI.1.3 : Code de base de la classe Point.

VI.2.2. Évaluation de la cohésion – ancienne approche

Dans le contexte de notre analyse, considérons la classe *Point* décrite par le code de la figure VI.1.2. Selon les critères de cohésion définis dans la section II.2, deux méthodes $\mathbf{M_i}$ et $\mathbf{M_j}$ peuvent être connectées de différentes manières : Elles partagent au moins une variable d'instance en commun (relation UA : usage d'attributs), ou interagissent au moins avec une méthode de la même classe (relation IM invocation de méthodes), ou partagent au moins un objet passé comme argument (relation CO paramètres objets en commun). Elles peuvent être connectées aussi indirectement. Donc, deux méthodes peuvent être directement ou indirectement connectées par un ou plusieurs critères.

➔ $UAMi \cap UAMj \neq \emptyset$ ou $IMMi \cap IMMj \neq \emptyset$ ou $UCOMi \cap UCOMj \neq \emptyset$.

Le graphe non dirigé G_I correspondant à cette classe est illustré par la figure VI.1.4.

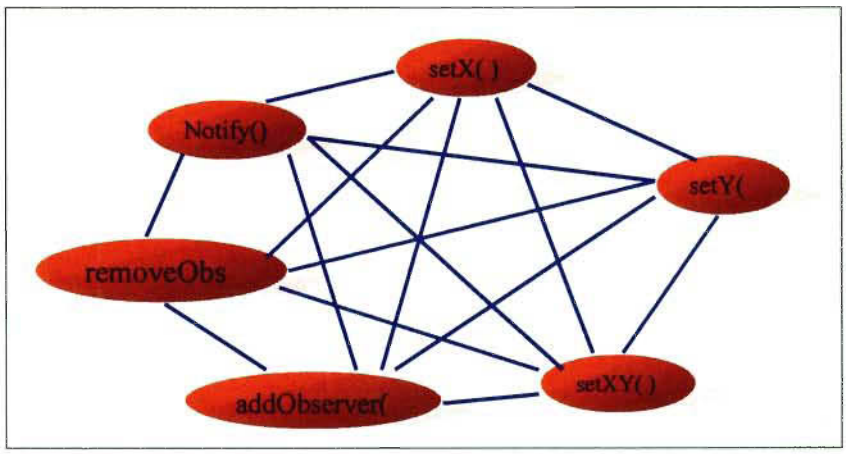

Figure VI.1.4 : Graphe non-dirigé G_I correspondant à la classe *Point*.

Le nombre des méthodes du graphe est : N=6;

Le nombre total de paires de méthodes du graphe est : Nmax=N*(N-1)/2 =15;

Le nombre de paires de méthodes connectées est : NC=15;

Par conséquent, DC_{IE}= 15/15 =1 ;

Ceci indique une cohésion totale de la classe *Point*. Si on se fie à cette approche, la classe est totalement cohésive et elle n'a pas besoin d'être restructurée. Ce qui n'est pas le cas comme nous l'avions déjà expliqué lors de la présentation de l'exemple. La classe comporte un enchevêtrement de code important dû au mixage avec le code relatif au mécanisme de notification des observateurs, et a été dotée de plus d'une responsabilité. En considérant juste les liens structurels, cette approche indique une cohésion parfaite et ne reflète pas exactement et fidèlement la réalité conceptuelle voir sémantique de la classe. La classe effectivement englobe du code correspondant à deux aspects (d'un point de vue conceptuel et sémantique) non liés.

VI.2.3. Évaluation de la cohésion – nouvelle approche

Appliquons maintenant notre nouvelle approche pour calculer la cohésion en se basant sur la classification, toujours avec la même classe *Point* de la figure VI.1.2. Son graphe de relations d'usage est donné par la figure VI.1.5.

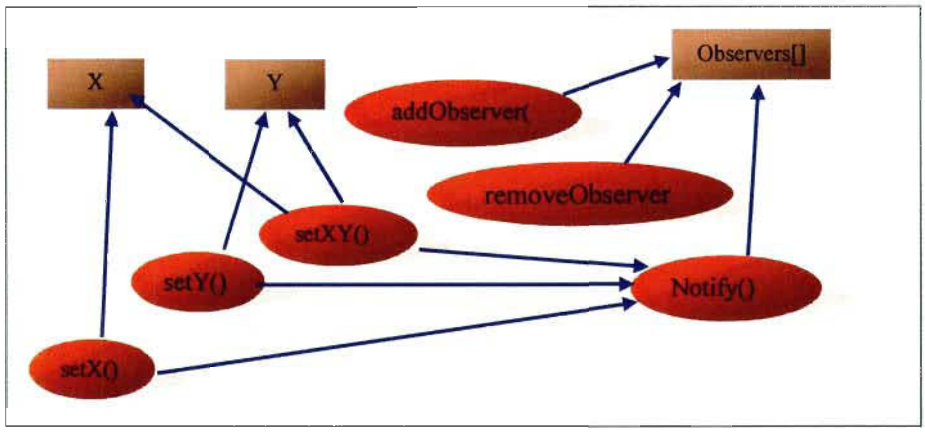

Figure VI.1.5 : Relation d'usage entre les membres de la classe *Point*.

Le Tableau VI.1.1 représente La matrice entités-propriétés, qui est obtenue selon les spécifications du modèle défini à la section (V.2.1). Elle a donc pour lignes les entités à classifier qui sont tous les attributs et les méthodes membres de la classe analysée, et pour colonnes les propriétés qui caractérisent chaque entité à classifier c.-à-d. les variables d'instances et les méthodes utilisées par les méthodes de cette classe.

Par exemple, comme la méthode *setX()* utilise l'attribut x, nous avons mis 1 dans la case (*setX() : x*) de cette matrice.

	propriétés								
Entités	***X***	***Y***	***observers***	***setX***	***setY***	***setXY***	***notify***	***Add Observer***	***Remove Observer***
setX	1	0	0	0	0	0	1	0	0
setY	0	1	0	0	0	0	1	0	0
setXY	1	1	0	0	0	0	1	0	0
Add Observer	0	0	1	0	0	0	0	0	0
Remove Observer	0	0	1	0	0	0	0	0	0
notify	0	0	1	0	0	0	0	0	0
X	1	0	0	1	0	1	0	0	0
Y	0	1	0	0	1	1	0	0	0
observers	0	0	1	0	0	0	1	1	1

Tableau VI.1.1 : Matrice entités-propriétés de la classe *Point*.

Afin d'appliquer l'algorithme de classification, nous avons choisi le **coefficient de Jaccard** comme indice de similarité. Cette matrice ainsi que l'indice de similarité sont fournis comme entrée pour l'outil XLSTAT, qui applique à son tour l'algorithme de classification intégré.

Pour obtenir la matrice des proximités, il commence par calculer la dissimilarité entre toute paire d'objets. Ici, les objets sont les méthodes $\boldsymbol{m_i}$ et les attributs $\boldsymbol{i_k}$ et le nombre

total d'entités à classifier est ***N=9***. La matrice des dissimilarités se déduit selon la formule : dissimilarité =1 – similarité. Par exemple, si on prend le cas de *setX()* et *setY()*, pour calculer la valeur de proximité entre ces deux méthodes selon l'indice de Jaccard (voir Tableau III.1 de la section III.3.3) :

La similarité S(*setX()* , *setY()*) = a/ (a + b + c)= 1/(1+1+1) = 1/3 = 0.333 ➔
Par conséquent, la dissimilarité d(*setX()* , *setY()*) =1 - 0.333 = 0.667.

Enfin, la matrice des proximités (Tableau VI.1.2) selon le coefficient de Jaccard (lien moyen) qui en résulte donne le degré de dissimilarité entre toute paire d'entités.

	setX	*setY*	*setXY*	*Add Observer*	*Remove Observer*	*notify*	*X*	*Y*	*observers*
setX	0,000	0,667	0,333	1,000	1,000	1,000	0,750	1,000	0,800
setY	0,667	0,000	0,333	1,000	1,000	1,000	1,000	0,750	0,800
setXY	0,333	0,333	0,000	1,000	1,000	1,000	0,800	0,800	0,833
Add Observer	1,000	1,000	1,000	0,000	0,000	0,000	1,000	1,000	0,750
Remove Observer	1,000	1,000	1,000	0,000	0,000	0,000	1,000	1,000	0,750
notify	1,000	1,000	1,000	0,000	0,000	0,000	1,000	1,000	0,750
X	0,750	1,000	0,800	1,000	1,000	1,000	0,000	0,800	1,000
Y	1,000	0,750	0,800	1,000	1,000	1,000	0,800	0,000	1,000
observers	0,800	0,800	0,833	0,750	0,750	0,750	1,000	1,000	0,000

Tableau VI.1.2 : Matrice des proximités correspondante à la classe *Point*.

L'algorithme est récursif. La fusion des clusters se fait récursivement, à chaque itération on fusionne les deux clusters les plus similaires (proches) et ainsi de suite jusqu'à aboutir à un seul cluster. À chaque fusion de deux clusters on crée un nœud. Le tableau VI.1.3 donne les statistiques des nœuds pour les agrégations successives.

Noeud	*Niveau*	*Poids*	*Objets*	*Fils gauche*	*Fils droit*
17	0,972	9	9	16	14
16	0,850	5	5	13	15
15	0,800	2	2	7	8
14	0,750	4	4	11	9
13	0,500	3	3	2	12
12	0,333	2	2	1	3
11	0,000	1	1	0	0
10	0,000	1	1	0	0

Tableau VI.1.3 : Statistiques des nœuds (la classe *Point*).

Sur ce tableau on peut noter le plus bas niveau de dissimilarité entre les membres de la classe ***Point*** qui est à son minimum (0). Il est marqué par les nœuds 10 et 11 dans cet exemple. Tandis que le niveau de dissimilarité le plus élevé entre les membres de la classe est égal à 0.972. Il est représenté par le nœud 17 résultant de la dernière fusion.

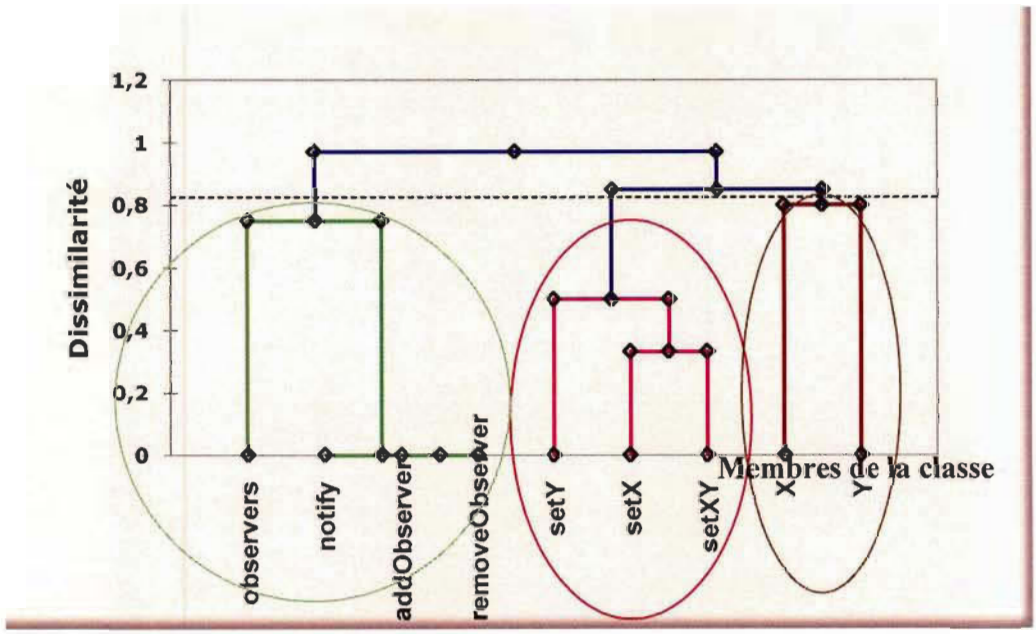

Figure VI.1.6 : Arbre hiérarchique correspondant à la classe *Point*.

L'arbre hiérarchique résultant est indiqué par la figure VI.1.6. Il est, en fait, la représentation graphique des nœuds résultants des fusions successives présentées précédemment dans le tableau des statistiques des nœuds.

Il faut noter que le choix du seuil de troncature établi par XLSTAT est basé sur la notion d'entropie décrite à la section III.5. Selon le niveau de troncature automatique déterminé par l'outil XLSTAT de l'arbre hiérarchique de la figure VI.1.6, on obtient 3 clusters :

1. {*observers[], notify(),addObserver(),removeObserver*()},
2. {*setY(),setX,setXY*()}
3. {*x,y*}.

Mais, du point de vue répartition des méthodes et leurs connectivités, on aura les deux clusters suivants :

1. K1={ *setY(), setX, setXY()* }
2. K2={ *notify(), addObserver(), removeObserver()* }

On a les 3 méthodes (*notify (), addObserver () et removeObserver*()) qui sont regroupées dans un cluster d'une part, et d'autre part nous avons les autres méthodes (*setY (), setXY() et setX*()) qui sont regroupées dans un second cluster (sous-arbre entouré d'un cercle en rouge).

Ce résultat révèle une information très importante sur la composition de la classe *Point.* En effet, on peut facilement déduire le nombre de composantes connexes qui est égal à 2. Cette information n'était pas déductible de l'ancienne approche de cohésion, mais la nouvelle approche est capable de la détecter. On peut aussi déduire le graphe de

connexions pour calculer la cohésion par le moyen de la classification. Deux méthodes qui sont classées dans le même cluster sont similaires et donc reliées, d'où le graphe de connexions de la figure VI.1.7.

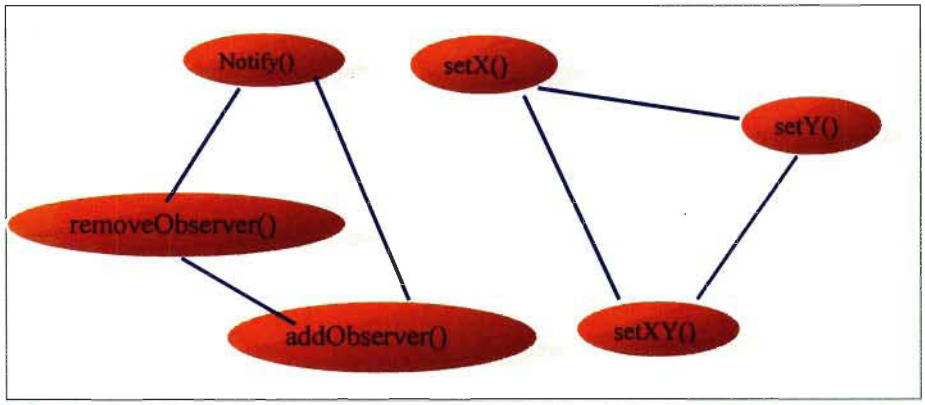

Figure VI.1.7 : Nouveau graphe de connexion G de la classe *Point*.

Et par conséquent :

• Nombre de composantes connexes NCC=2.

Le nombre de méthodes du graphe est : N=6 ;

Le nombre total de paires de méthodes est : Nmax=N*(N-1)/2 =15 ;

Le nombre de paires de méthodes connectées NC=6 ;

Par conséquent, COH_{CL}= NC/Nmax = 0.4.

Ce qui indique une relative faible cohésion pour la classe *Point*.

VI.2.4. Discussion

Une valeur faible de la métrique COH_{CL} (0.4) indique d'une manière générale que les membres de la classe sont faiblement reliés entre eux. Bien qu'ils puissent constituer un

seul groupe représentant un comportement bien défini. Toutefois, elle peut aussi implicitement indiquer l'existence de plusieurs groupes connexes ou d'une disparité dans le code; NCC vaut 2. Ce résultat signifie que la classe comporte deux groupes disjoints, donc séparables en l'occurrence **k1 et k2**.

Ces résultats (COH_{CL} et NCC) reflètent parfaitement l'état de la classe (figure VI.1.2); deux responsabilités ou rôles (son rôle principal comme point, et un rôle secondaire relatif à la gestion du mécanisme de notification pour ses objets observateurs). Malgré l'entremêlement et la dispersion du code relatifs à ces deux rôles, la nouvelle approche a pu capturer et isoler les éléments reliés de façon appropriée. D'ailleurs, on note selon la décomposition des clusters k1 et k2, que chaque cluster contient uniquement les méthodes propres à une même responsabilité. Le cluster k1 a regroupé les éléments de code A (figure VI.1.2), tandis que le cluster k2 a regroupé les éléments de code B et C (figure VI.1.2).

Les valeurs de cohésion selon la nouvelle approche reflètent mieux le fait que la classe peut être considérée comme candidate pour une restructuration. Ce qui n'était pas déductible avec l'ancienne approche de cohésion.

Une des problématiques liées à l'approche objet est la prise en charge des préoccupations transverses. Malgré l'utilisation du pattern observateur, la classe Point se voit affectée deux rôles disparates. La programmation orientée aspect a pour but de palier aux limites de la programmation orientée objet. Son principal apport est de regrouper les différentes préoccupations transversales d'un système (journalisation, persistance) dans des modules appelés aspect. Grâce à ce nouveau paradigme, on pourrait facilement isoler dans un aspect à part le code relatif à la gestion et notification des objets observateurs du reste du code propre à l'objet observé (le sujet :

dans notre exemple *Point*) d'après la solution proposée par Hannemann et Kiczales [Hannemann 02], (figures IV.1.8 et IV.1.9).

Les auteurs ont défini de façon abstraite un aspect (*ObserverProtocole*) illustré par la figure VI.1.8, pour l'implémentation générique du pattern. Ils ont défini les points de coupure (pointcuts) pour désigner les endroits où le pattern doit être intégré dans le programme et les conditions sous lesquelles il doit être déclenché.

```
01 public abstract aspect ObserverProtocol {
02
03   protected interface Subject   { }
04   protected interface Observer { }
05
06   private WeakHashMap perSubjectObservers;07
08   protected List getObservers(Subject s) {
09     if (perSubjectObservers == null) {
10       perSubjectObservers = new WeakHashMap();
11     }
12     List observers =
13        (List)perSubjectObservers.get(s);
14     if ( observers == null ) {
15       observers = new LinkedList();
16       perSubjectObservers.put(s, observers);
17     }
18     return observers;
19   }
20
21   public void addObserver(Subject s,Observer o){
22     getObservers(s).add(o);
23   }
24   public void removeObserver(Subject s,Observer o){
25     getObservers(s).remove(o);
26   }
27
28   abstract protected pointcut
29     subjectChange(Subject s);
30
31   abstract protected void
32     updateObserver(Subject s, Observer o);
33
34   after(Subject s): subjectChange(s) {
35     Iterator iter = getObservers(s).iterator();
36     while ( iter.hasNext() ) {
37       updateObserver(s, ((Observer)iter.next()));
38     }
39   }
40 }
```

Figure VI.1.8 : Aspect généralisé ObserverProtocol.

Cet aspect a été ensuite étendu pour définir le code relatif à un aspect spécifique. Comme ici, on a défini deux aspects spécifiques (figure VI.1.9) comme suit:

1. ***ColorObserver*** : relatif à l'aspect changement de couleur du sujet
2. ***CoordinateObserver*** : relatif à l'aspect changement de position (de coordonnées), celui-ci déclare un point de jonction qui représente l'appel à n'importe quelle méthode préfixée par set et appartenant soit à un objet de type *Point* soit à un objet de type *Line*. Ce point de jonction est associé à un greffon (advice) qui a pour fonction d'informer l'objet Observateur du changement.

```
01 public aspect ColorObserver extends ObserverProtocol {
02
03   declare parents: Point  implements Subject;
04   declare parents: Line   implements Subject;
05   declare parents: Screen implements Observer;
06
07   protected pointcut subjectChange(Subject s):
08     (call(void Point.setColor(Color)) ||
09      call(void  Line.setColor(Color)) ) && target(s);
10
11   protected void updateObserver(Subject s,
12                                 Observer o) {
13     ((Screen)o).display("Color change.");
14   }
15 }
```

```
16 public aspect CoordinateObserver extends
17   ObserverProtocol {
18
19   declare parents: Point  implements Subject;
20   declare parents: Line   implements Subject;
21   declare parents: Screen implements Observer;
22
23   protected pointcut subjectChange(Subject s):
24     (call(void Point.setX(int))
25      || call(void Point.setY(int))
26      || call(void Line.setP1(Point))
27      || call(void Line.setP2(Point)) ) && target(s);
28
29   protected void updateObserver(Subject s,
30                                 Observer o) {
31     ((Screen)o).display("Coordinate change.");
32   }
33 }
```

Figure VI.1.9 : Deux instances Observers différentes.

Cette solution orientée aspect est de loin plus intéressante et meilleure que celle proposée par la programmation OO :

- élimination des préoccupations transverses (scattering et tangling),
- code mieux localisé, réutilisé, plus compréhensible,
- l'implémentation a laissé le code de base du programme intact (inchangé).

En effet, le code de base de notre programme, par exemple, dans le cas de la classe *Point* se réduira par conséquent à celui de la figure VI.1.3.

On constate qu'avec cette nouvelle approche d'évaluation de la cohésion, nous avons été capables de déceler une disparité dans le code dont la raison est liée à la présence dans la classe de code correspondant à des préoccupations transverses. Ceci représente, selon nous, un cas particulier des symptômes de défauts de conception qu'on peut traiter avec notre approche. Ceci constitue une piste intéressante qu'il serait pertinent d'explorer.

VI.3. Exemple 2 (la classe TangledStack)

VI.3.1. Présentation de l'exemple

C'est un exemple tiré de [Miguel 05]. Il présente une classe encapsulant les opérations de gestion d'une pile (fonctionnalité principale : empiler, dépiler, tester si elle est vide ou pleine) en plus de celles reliée à une fonctionnalité secondaire qui lui a été affectée, qui est l'affichage de son contenu dans un champ de texte d'une fenêtre (frame).

Le code est donné par le listing de la figure VI.2.1. Les parties du code surlignées en jaune (bout de code étiqueté **A** sur la figure VI.2.1) sont relatives à la fonctionnalité secondaire.

Le fait de doter cette classe d'une seconde responsabilité (affichage du contenu), a rendu le code de base enchevêtré et ambiguë. Contrairement à ce qu'il devrait être en principe.

```
public class TangledStack {
 private int _top = -1;
 private Object[] _elements;
 private final int S_SIZE = 10;
 private JLabel _label = new JLabel("Stack ");
 private JTextField _text = new JTextField(20);

 public TangledStack(JFrame frame) {
  _elements = new Object[S_SIZE];
  frame.getContentPane().add(_label);
   text.setText("[]");
  frame.getContentPane().add(_text);
 }
 public String toString() {
  StringBuffer result = new StringBuffer("[");
  for (int i = 0; i <= _top; i++) {
   result.append(_elements[i].toString());
   if (i != _top)
    result.append(", ");
  }
  result.append("]");
  return result.toString();
 }
 private void display() {
  _text.setText(toString());
 }

 public void push(Object element) throws PreConditionException {
  if (isFull())
   throw new PreConditionException("push when stack full.");
  _elements[++_top] = element;
  display();
 }

 public void pop() throws PreConditionException {
  if (isEmpty())
   throw new PreConditionException("pop when stack empty.");
   top--;
  display();
 }
 public Object top() throws PreConditionException {
  if (isEmpty())
   throw new PreConditionException("top when stack empty.");
  return _elements[_top];
 }
 public boolean isFull() {
  return (_top == S_SIZE - 1);
 }
 public boolean isEmpty() {
  return (_top < 0); }
}
```

A

Figure VI.2.1 : Listing du code de la classe *TangledStack*.

VI.3.2. Évaluation de la cohésion – ancienne approche

Selon les critères de cohésion étendue décrits dans la section II.2, nous obtenons le graphe non dirigé G_I correspondant à cette classe donné par le schéma de la figure VI.2.2.

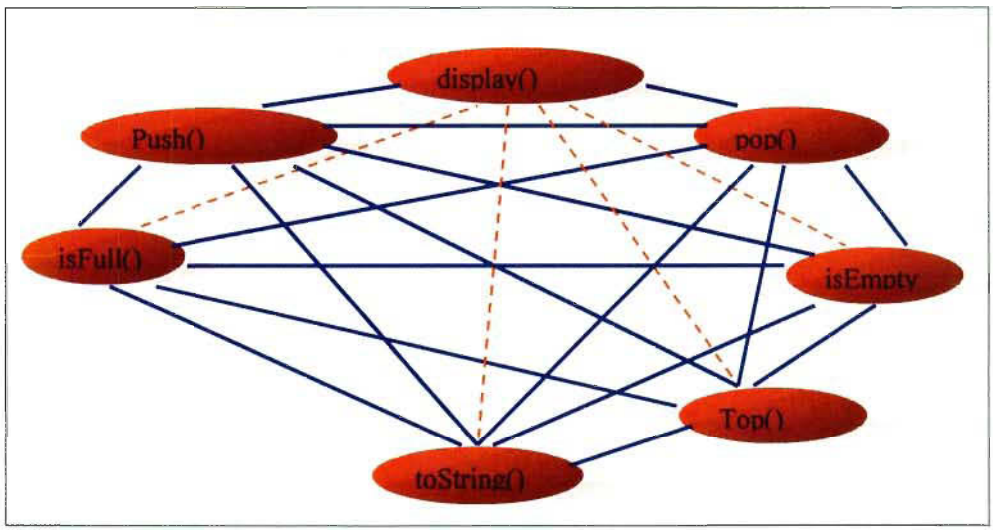

Figure VI.2.2 : Graphe non dirigé G_I correspondant à la classe *TangledStack*.

Le nombre de méthodes dans le graphe est : N=7;
Le nombre total de paires de méthodes du graphe est : Nmax= N*(N-1)/2 =21;
Le nombre de paires de méthodes connectées est : NC = 21-4 = 17;

Par conséquent, DC_{IE}=17/21 =0.80.

Ce qui indique une cohésion qui est quand même assez élevée. La classe peut donc (selon ce résultat) être considérée comme acceptable du point de vue conception.

VI.3.3. Évaluation de la cohésion – nouvelle approche

Appliquons la nouvelle approche de cohésion sur la même classe *TangledStack*. Le graphe de la figure VI.2.3 représente son graphe de relations d'usage.

La matrice entités-propriétés (Tableau VI.2.1) est obtenue selon les spécifications du modèle défini à la section V.2.1.

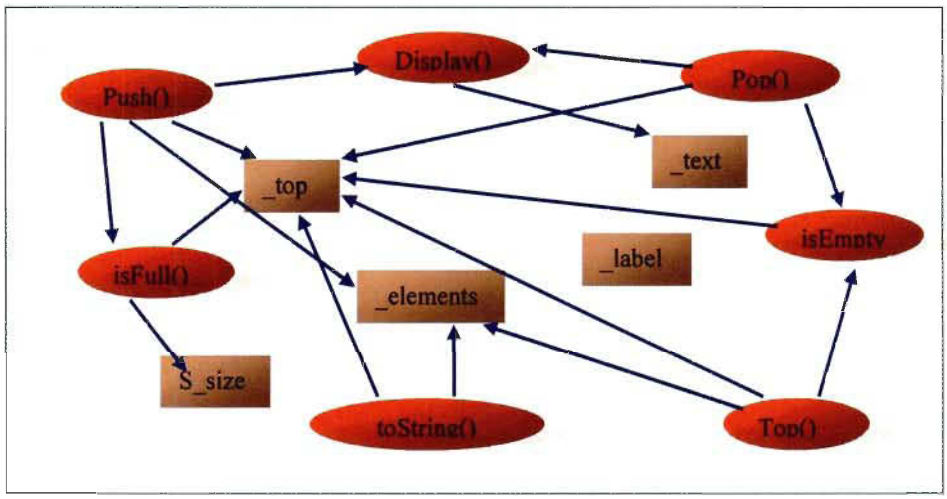

Figure VI.2.3 : Relation d'usage entre les membres de la classe *TangledStack*.

La matrice des proximités selon le coefficient de Jaccard donnant le degré de dissimilarité entre toutes paires d'entités, le tableau des statistiques des nœuds ainsi que l'arbre hiérarchique résultant de la procédure de classification hiérarchique sont donnés respectivement par les tableaux IV.2.2 et IV.2.3 ainsi que la figure VI.2.4.

	propriétés											
Entités	***Attributs***					***méthodes***						
	_top	*_element*	*_label*	*_text*	*S_size*	*m1*	*m2*	*m3*	*m4*	*m5*	*m6*	*m7*
m1:display()	0	0	0	1	0	0	0	0	0	0	0	0
m2:push()	1	1	0	0	0	1	0	0	0	1	0	0
m3:pop()	1	0	0	0	0	1	0	0	0	0	1	0
m4:top()	1	1	0	0	0	0	0	0	0	0	1	0
m5:isFull()	1	0	0	0	1	0	0	0	0	0	0	0
m6:isEmpty()	1	0	0	0	0	0	0	0	0	0	0	0
m7:toString()	1	1	0	0	0	0	0	0	0	0	0	0
_top	1	0	0	0	0	0	1	1	1	1	1	1
_element	0	1	0	0	0	0	1	0	1	0	0	1
_label	0	0	1	0	0	0	0	0	0	0	0	0
_text	0	0	0	1	0	1	0	0	0	0	0	0
S_size	0	0	0	0	1	0	0	0	0	1	0	0

Tableau VI.2.1 : Matrice entités-propriétés de la classe *TangledStack*.

	display()	***push()***	***pop()***	***top()***	***isFull()***	***isEmpty()***	***toString()***	***_top***	***_element***	***_label***	***_text***	***S_size***
display()	0,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,50	1,00
push()	1,00	0,00	0,60	0,60	0,80	0,75	0,50	0,78	0,86	1,00	0,80	0,80
pop()	1,00	0,60	0,00	0,50	0,75	0,67	0,75	0,75	1,00	1,00	0,75	1,00
top()	1,00	0,60	0,50	0,00	0,75	0,67	0,33	0,75	0,83	1,00	1,00	1,00
isFull()	1,00	0,80	0,75	0,75	0,00	0,50	0,67	0,88	1,00	1,00	1,00	0,67
isEmpty()	1,00	0,75	0,67	0,67	0,50	0,00	0,50	0,86	1,00	1,00	1,00	1,00
toString()	1,00	0,50	0,75	0,33	0,67	0,50	0,00	0,88	0,80	1,00	1,00	1,00
_top	1,00	0,78	0,75	0,75	0,88	0,86	0,88	0,00	0,63	1,00	1,00	0,88
_element	1,00	0,86	1,00	0,83	1,00	1,00	0,80	0,63	0,00	1,00	1,00	1,00
_label	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,00	1,00	1,00
_text	0,50	0,80	0,75	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,00	1,00
S_size	1,00	0,80	1,00	1,00	0,67	1,00	1,00	0,88	1,00	1,00	1,00	0,00

Tableau VI.2.2 : Matrice des proximités de la classe *TangledStack*.

Nœud	*Niveau*	*Poids*	*Objets*	*Fils gauche*	*Fils droit*
23	1,000	12	12	10	22
22	0,975	11	11	14	21
21	0,918	9	9	20	12
20	0,865	8	8	19	18
19	0,694	6	6	15	17
18	0,625	2	2	8	9
17	0,617	4	4	3	16
16	0,550	3	3	2	13
15	0,500	2	2	1	11
14	0,500	2	2	5	6
13	0,333	2	2	4	7

Tableau VI.2.3 : Statistiques des nœuds (la classe *TangledStack*).

On note que le plus bas niveau de dissimilarité entre les membres de la classe ***TangledStack*** est 0.333. Il est marqué par le nœud 13. Tandis que le niveau de dissimilarité le plus élevé entre les membres de cette classe est à son maximum (1). Il est représenté par le nœud 23.

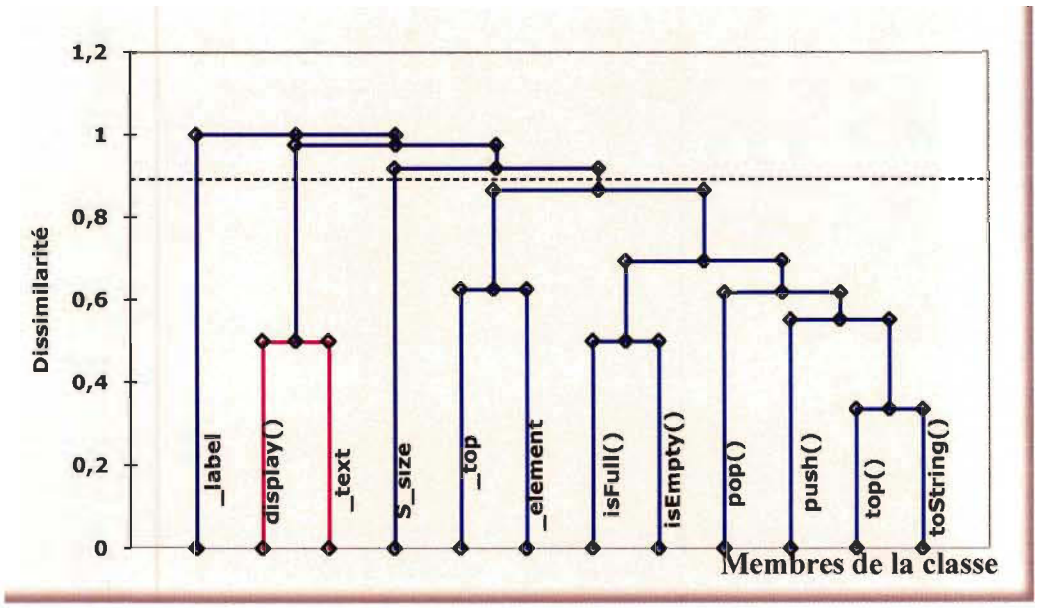

Figure VI.2.4 : L'arbre hiérarchique de la classe *TangledStack*.

Selon le niveau de troncature automatique (XLSTAT) de l'arbre hiérarchique (figure VI.2.4), on obtient 4 clusters, mais du point de vue répartition des méthodes et leur connectivité on aura les deux clusters suivants :

1. K1={ *isEmpty(), isFull(), pop(), push(), top(), toString()* }
2. K2={ *display()* }

Deux méthodes qui sont classées dans le même cluster sont similaires et donc reliées, d'où le graphe de connexions de la figure VI.2.5.

Par conséquent :
Le nombre de composantes connexes est : NCC=2.

Le nombre de méthodes du graphe est : N=7;
Le nombre total des paires de méthodes est : Nmax =N*(N-1)/2 =21;
Le nombre de paires de méthodes connectées est : NC=15;

Par conséquent, COH_{CL}= NC/Nmax = 0.71.

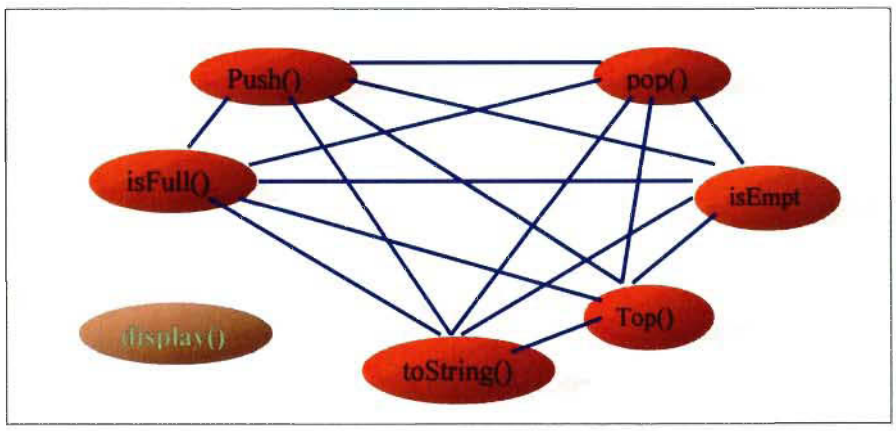

Figure VI.2.5 : Nouveau graphe de connexion G de la classe *TangledStack*.

VI.3.4. Discussion

La valeur de la métrique COH_{CL} (0.71) n'est pas aussi élevée que la valeur fournie par l'ancienne approche. Prise seule, elle ne peut rien indiquer sur l'état global de la classe. Toutefois, puisque la métrique NCC a une valeur de 2, alors cela signifie que la classe comporte deux groupes disjoints, en l'occurrence **k1 et k2**. En effet, la classe pile possède deux rôles distincts; son rôle principal comme pile standard, et un rôle secondaire relatif au mécanisme d'affichage dans une interface graphique.

La nouvelle approche a pu isoler les éléments reliés à chaque préoccupation de façon appropriée. D'ailleurs, on note selon la décomposition des clusters k1 et k2, que chaque cluster contient uniquement les méthodes relatives à une même responsabilité. Le cluster k2 a regroupé les éléments de code A (figure VI.2.1), tandis que le cluster k1 a regroupé les éléments de code de base de la classe *Pile* standard.

Les valeurs 0.71 et 2 correspondantes aux deux métriques COH_{CL} et NCC respectivement, selon la nouvelle approche donnent une meilleure explication et démontrent que la classe est une bonne candidate pour la restructuration. Ce qui n'était pas le cas avec l'ancienne approche.

Dans le cas de cet exemple, il s'agit aussi d'une préoccupation transverse qui se recoupe avec la principale préoccupation de la classe Pile. Une solution orientée-aspect peut rendre le code de cette classe meilleure en termes de structure et de compréhension en factorisant le code secondaire relatif à l'affichage du contenu de la pile dans un aspect (figure VI.2.7) à part, laissant ainsi le code de base de la classe plus clair et concentré sur la fonctionnalité principale de gestion d'une pile standard (figure VI.2.6).

```
public class Stack {

 private int _top = -1;
 private Object[] _elements;
 private final int S_SIZE = 10;

 public TangledStack() {
  _elements = new Object[S_SIZE];

 }

 public String toString() {
  StringBuffer result = new StringBuffer("[");
  for (int i = 0; i <= _top; i++) {
   result.append(_elements[i].toString());
   if (i != _top)
    result.append(", ");
  }
  result.append("]");
  return result.toString();
 }


public void push(Object element) throws PreConditionException {
  if (isFull())
   throw new PreConditionException("push when stack full.");
  _elements[++_top] = element;
}

 public void pop() throws PreConditionException {
  if (isEmpty())
   throw new PreConditionException("pop when stack empty.");
  _top--;
}

 public Object top() throws PreConditionException {
  if (isEmpty())
   throw new PreConditionException("top when stack empty.");
  return _elements[_top];
 }

 public boolean isFull() {
  return (_top == S_SIZE - 1);
 }
 public boolean isEmpty() {
  return (_top < 0);
 }
}
```

Figure VI.2.6 : Listing du code de la classe *TangledStack* (code enchevêtré enlevé).

```
public aspect WindowView {
      private JLabel TangledStack._label = new JLabel("Stack "),
      private JTextField TangledStack._text = new JTextField(20);

      public TangledStack.new(JFrame frame) {
            this();

            frame.getContentPane().add(_label);
            _text.setText("[]");
            frame.getContentPane().add(_text);

      }

      private void TangledStack.display() {
            _text.setText(toString());
      }

 pointcut stateChange(TangledStack stack):
    (execution(public void stack.TangledStack.push(Object))
                ||
    execution(public void stack.TangledStack.pop())) && this(stack);

after(TangledStack _this) returning :

stateChange(_this) {
            _this.display();
      }
}
```

Figure VI.2.7 : Code de l'aspect « affichage » pour la classe *TangledStack*..

Encore une fois, nous constatons la capacité de notre approche à déceler les cas de présence de symptômes de code transverse et à l'identifier de façon efficace. Ce qui n'est pas envisageable avec les anciennes approches de cohésion.

VI.4. Exemple 3 (Design pattern Chaîne de responsabilité)

VI.4.1. Présentation de l'exemple

Cet exemple est tiré d'une expérimentation réalisée par Cooper [Cooper 00]. Il a également été repris par Miguel [Miguel 05]. Idéalement, chaque classe doit assurer un seul rôle (contenir un ensemble cohérent de responsabilités). Malheureusement, ce n'est toujours pas le cas. L'implémentation orientée-objet des patrons de conception mènent généralement à cette situation comme c'est aussi le cas des rôles sur- imposés (« superimposed roles ») tels que désignés par Hannemann et Kiczales [Hannemann 02].

Les interfaces sont une manière populaire pour modéliser les rôles en Java. L'implémentation des interfaces est un symptôme utile qui peut aider à détecter une double « personnalité » dans le code source Java. Quand une classe implémente une interface modélisant un rôle qui n'est pas relié à la préoccupation primaire de la classe, alors la classe est dite porteuse du « code smell » Double Personnalité [Miguel 05].

Le patron de conception Chaîne de responsabilité (figure VI.3.1) permet à un nombre quelconque de classes de tenter de répondre à une requête sans connaître les possibilités des autres classes sur cette requête, permettant ainsi de diminuer le couplage entre objets. Chaque fois qu'un objet reçoit un message dont il ne peut pas s'occuper, il le délègue à l'objet suivant dans la chaîne. Le seul lien commun entre ces objets (qui peuvent être de différents types) étant cette logique qu'ils devront tous avoir pour passer la requête entre eux jusqu'à ce que l'un des objets puisse répondre. Généralement, on a recours à ce patron lorsqu'on a une suite de processus ou d'algorithmes à utiliser, mais qu'on ne sait pas dans quel contexte les utiliser :

• Plus d'un objet peut saisir une requête, et ce dernier n'est pas connu à priori, il sera déterminé automatiquement selon le contexte;
• On veut émettre une requête à un objet parmi plusieurs sans spécifier le récepteur explicitement;
• L'ensemble des objets qui peuvent saisir la requête doit être spécifié dynamiquement.

On décide donc de créer un mécanisme pour qu'ils décident eux-mêmes s'ils correspondent à un contexte donné.

But

Créer une chaîne de processus, encapsulés dans des objets, et permettre de propager l'appel à ce processus dans la chaîne, en laissant le soin à l'objet/processus de décider si le contexte le concerne ou pas.

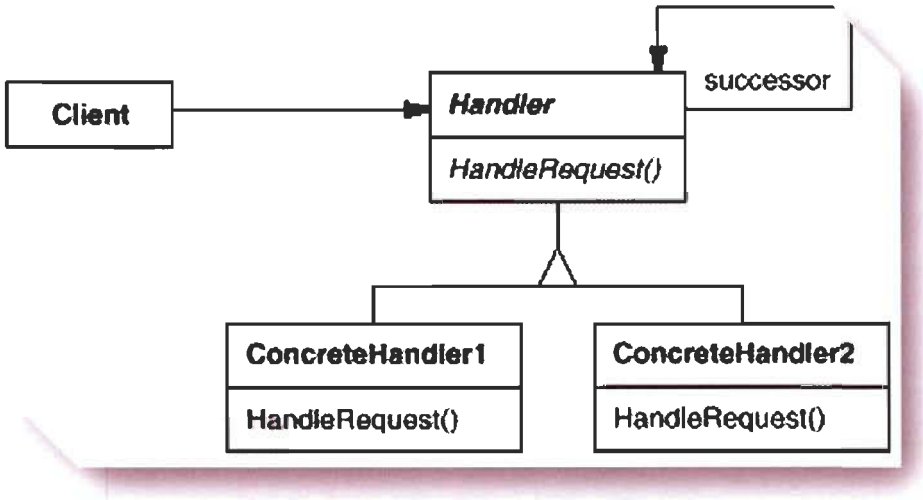

Figure VI.3.1 : Diagramme UML de la structure du patron de conception Chaîne de responsabilités.

Participants et implémentation

1. Les tâches de la classe *Handler,* consistent à:

- Définir une interface pour saisir les requêtes
- Implémenter le lien successeur
- Définir des méthodes de base permettant d'ajouter, de retirer, d'accéder à une chaîne (*AddNextHandler, removeNextHandler, getNextHandler*).

2. La classe *ConcreteHandler* (vérifie son éligibilité et appelle le chaînon suivant)

- S'occupe des requêtes dont elle est responsable
- Peut accéder à son successeur dans la chaîne
- S'il ne peut pas saisir la requête, il l'achemine vers son successeur.

3. Client

- Initier la requête vers un objet *ConcreteHandler* dans la chaîne.

La figure VI.3.2 montre une implémentation possible du patron chaîne de responsabilité par une classe *ColorImage.* Le rôle secondaire est modélisé par l'interface Chain, dont tous les objets participants doivent l'implémenter (le code relatif à ce rôle est ombré, désigné par une étiquette sur la figure VI.3.2).

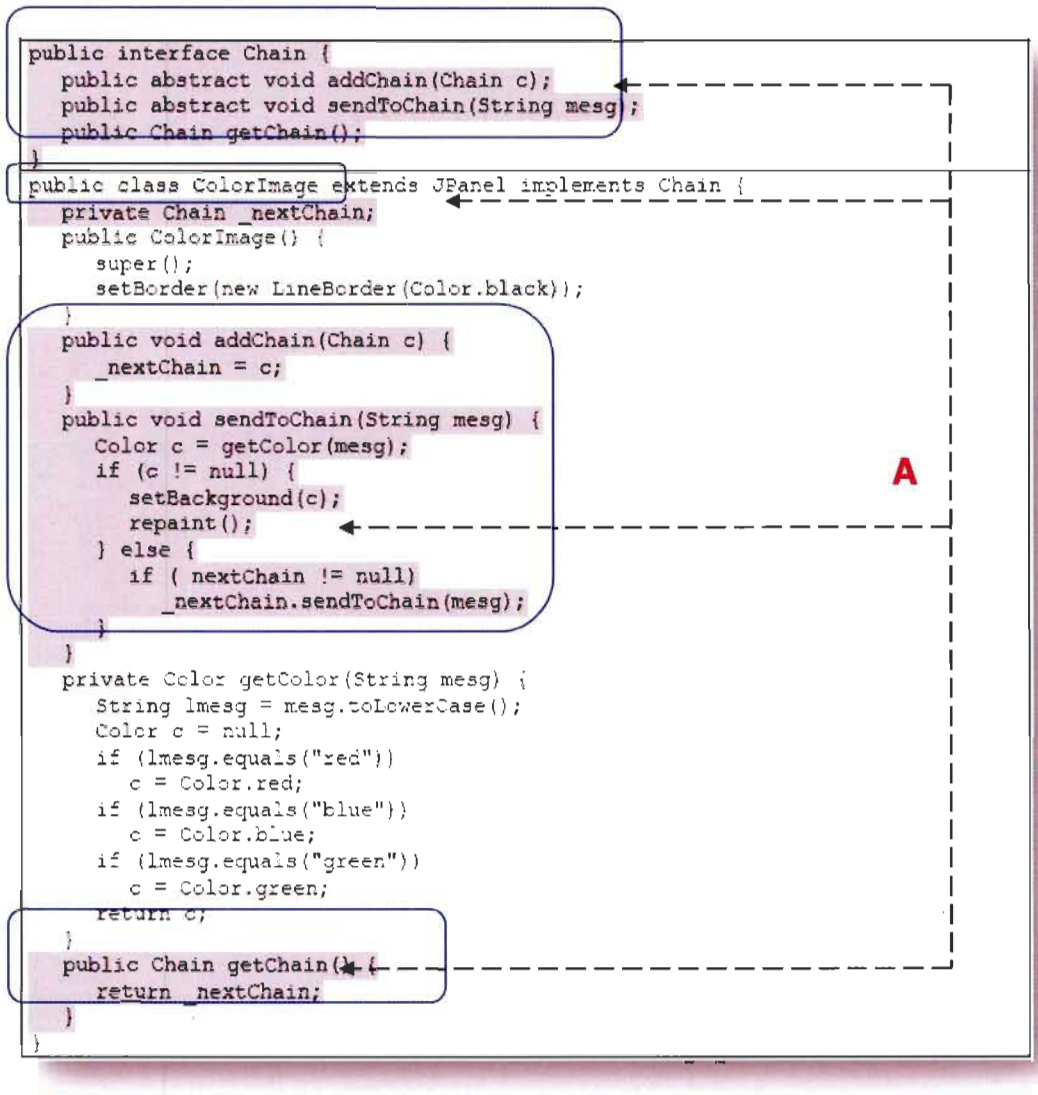

Figure VI.3.2 : Exemple d'implémentation du patron chaîne de responsabilités (la classe *ColorImage)*.

La méthode *addChain*() ajoute une autre classe à la chaîne des classes. La méthode *getChain()* retourne l'objet courant auquel les messages sont adressés. Ces deux méthodes nous permettent de modifier la chaîne dynamiquement et d'ajouter des classes additionnelles dans le milieu de la chaîne.

La méthode *sendToChain()* achemine un message à l'objet suivant dans la chaîne. On peut clairement distinguer l'enchevêtrement du code résultant de l'implémentation du mécanisme nécessaire pour gérer le message reçu par les objets de cette classe. En fait, on voit bien que la classe a été dotée d'une seconde responsabilité relative à la manipulation des messages reçus et leur passation aux objets suivants dans la chaîne d'objets participants au cas où l'objet en cours ne pourrait pas saisir le message. Sinon, la classe serait réduite à un code très simple concernant uniquement la manipulation de la couleur. Il y a 2 situations différentes:

1- un rôle secondaire surimposé à une seule classe

2- un rôle transverse coupant des classes multiples

Si des classes multiples implémentent l'interface, il s'agit alors d'une préoccupation transverse.

VI.4.2. Évaluation de la cohésion – ancienne approche

Considérons la classe *ColorImage* (figure VI.3.2). Selon les critères de cohésion étendue décrits dans les sections II.2 du chapitre II, nous obtenons le graphe non dirigé G_I correspondant à cette classe donné par le schéma de la figure VI.3.3:

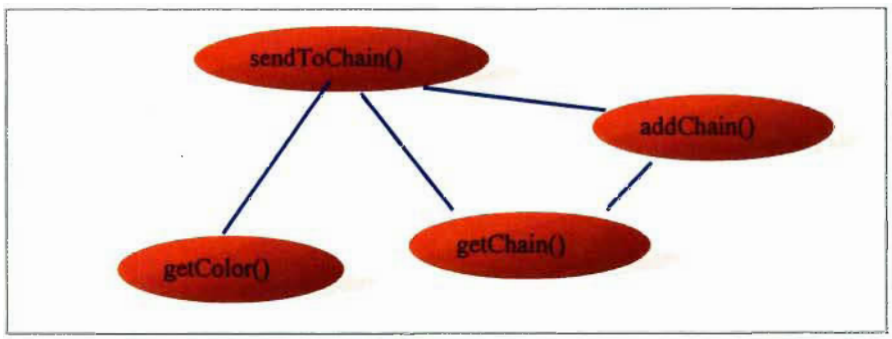

Figure VI.3.3 : Graphe non dirigé G_I correspondant à la classe *ColorImage*.

Le graphe contient quatre méthodes : N=4;

Le nombre total de paires de méthodes du graphe est : Nmax= N*(N-1)/2 =6;

Le nombre de paires de méthodes connectées est : NC=4 ;

Par conséquent, DC_{IE}= NC/Nmax =0.667.

Ce qui représente quand même une cohésion assez bonne pour la classe *ColorImage*.

VI.4.3. Évaluation de la cohésion – nouvelle approche

Appliquons la nouvelle approche pour l'évaluation de la cohésion sur la même classe *ColorImage*. Le graphe de la figure VI.3.4 représente son graphe de relations d'usage.

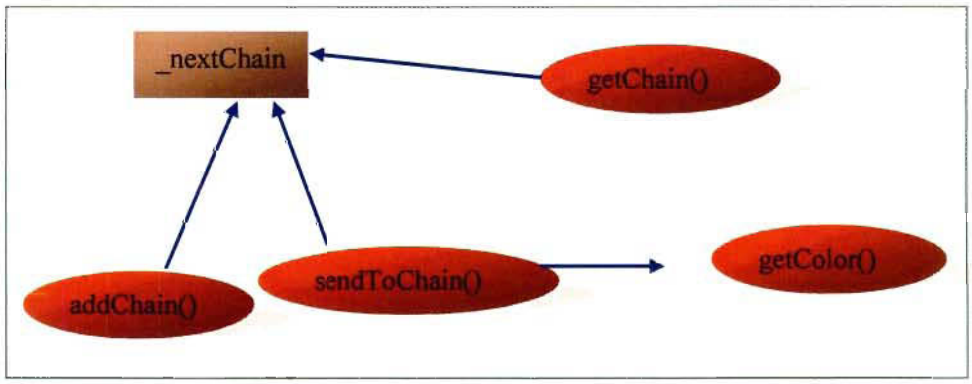

Figure VI.3.4 : Relation d'usage entre les membres de la classe *ColorImage*.

La matrice entités-propriétés (Tableau VI.3.1) est obtenue selon les spécifications du modèle défini à la section V.2.1.

	propriétés				
Entités	*next_Chain*	*addChain()*	*sendToChain()*	*getColor()*	*getChain()*
addChain()	1	0	0	0	0
sendTochain()	1	0	1	1	0
getColor()	0	0	0	0	0
getChain()	1	0	0	0	0
next_Chain	1	1	1	0	1

Tableau VI.3.1 : Matrice Entités-propriétés de la classe *ColorImage*.

La matrice des proximités selon le coefficient Jaccard, le tableau des statistiques des nœuds ainsi que l'arbre hiérarchique résultant de la procédure de classification hiérarchique sont donnés respectivement par les tableaux IV.3.2 et IV.3.3 et la figure VI.3.5.

	addChain()	***sendTochain()***	***getColor()***	***getChain()***	***next_Chain***
addChain()	0,000	0,667	1,000	0,000	0,750
sendTochain()	0,667	0,000	1,000	0,667	0,600
getColor()	1,000	1,000	0,000	1,000	1,000
getChain()	0,000	0,667	1,000	0,000	0,750
next_Chain	0,750	0,600	1,000	0,750	0,000

Tableau VI.3.2 : Matrice des proximités de la classe *ColorImage*.

Noeud	***Niveau***	***Poids***	***Objets***	***Fils gauche***	***Fils droit***
9	1,000	5	5	3	8
8	0,708	4	4	6	7
7	0,600	2	2	2	5
6	0,000	1	1	0	0

Tableau VI.3.3 : Statistiques des nœuds (la classe *ColorImage*).

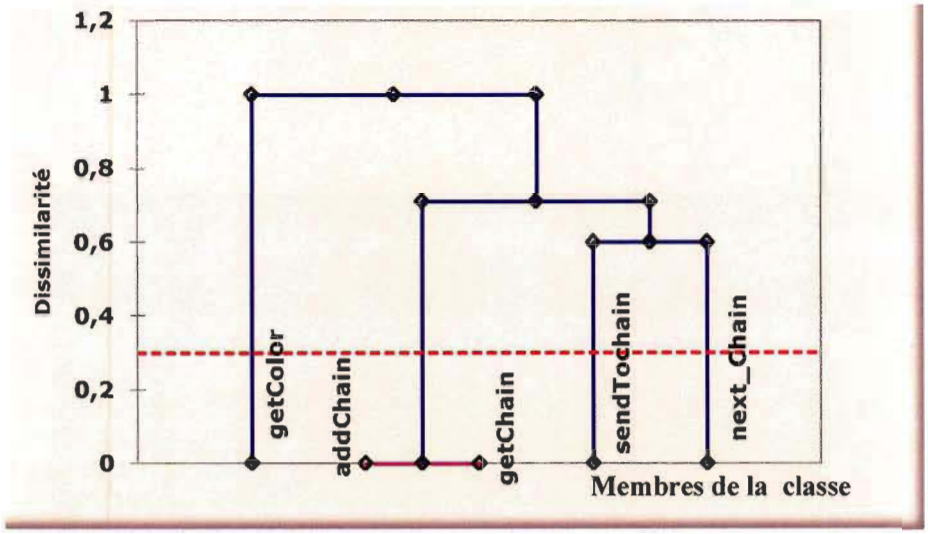

Figure VI.3.5 : Arbre hiérarchique correspondant à la classe *ColorImage*.

Selon le niveau de troncature automatique (XLSTAT) de l'arbre hiérarchique (figure VI.3.5), on aura 3 clusters **par rapport à la répartition des méthodes** comme suit :

1. K1={*addChain (), getChain ()*}
2. K2={*sendToChain()*}
3. K3={*getColor()*}.

La classe *ColorImage* a un nombre de composantes connexes égales à 3. Selon la classification, deux méthodes qui sont classées dans le même cluster sont similaires et donc reliées, d'où le graphe de connexions de la figure VI.3.6.

Le nombre de composantes connexes est : NCC=3;
Le nombre de méthodes du graphe est : N=4 ;
Le nombre total de paires de méthodes du graphe est : Nmax=N*(N-1)/2 =6;
Le nombre de paires de méthodes connectées est : NC=3 ;

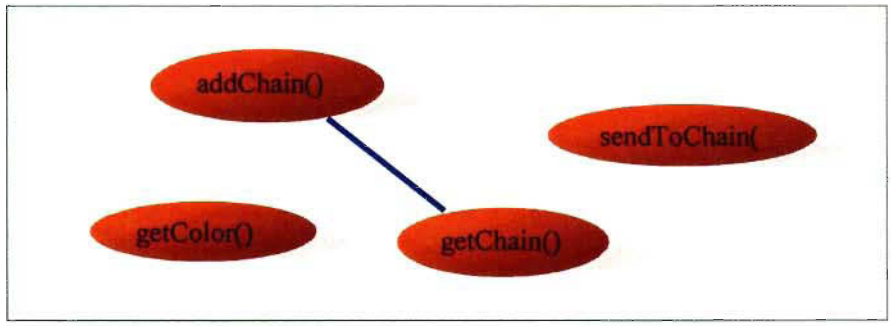

Figure VI.3.6 : Nouveau graphe de connexion G de la classe *ColorImage* selon la nouvelle approche.

Par conséquent, COH_{CL}= 1/6=0.16

Cela indique une très faible cohésion pour la classe *ColorImage*.

VI.4.4. Discussion

La métrique COH_{CL} indique une valeur de 0.16, ce qui est presque négligeable comme cohésion au sein de cette classe. Sans même considérer la valeur de l'autre métrique NCC, on peut affirmer que la classe doit être révisée pour une éventuelle restructuration. La valeur de NCC est égale à 3. Cependant, l'inspection manuelle du code indique que la classe possède deux rôles différents, et non pas trois (3). Dans ces conditions, on peut dire que nous avons pu déceler la disparité dans le code, sauf que l'évaluation a été un peu exagérée.

D'ailleurs, on note selon la décomposition des clusters k1, k2 et k3, que chaque cluster contient uniquement les méthodes propres à une même responsabilité. Les clusters k1 et k2 ont regroupé chacun les éléments de code A (figure VI.3.2), tandis que le cluster k3

a regroupé uniquement les éléments de code de base de la classe *ColorImage* qui est la méthode *getColor()* (cela signifie qu'il n'y a pas d'enchevêtrement de code dans un même cluster).

VI.5. Exemple 4 (Design pattern Singleton)

VI.5.1. Présentation de l'exemple

Parfois, il est nécessaire et suffisant d'avoir une seule instance d'une classe donnée pendant l'exécution d'une application (par exemple un seul objet de connexion de base de données). Le pattern singleton est alors très utile dans ce cas, puisqu'il permet d'assurer l'existence d'une et une seule instance d'un objet particulier. Mais, qui doit être le responsable pour maintenir cette tâche?

Déclarer une variable globale pour maintenir l'instance de cette classe semble facile, mais cela n'empêche pas les objets clients de créer d'autres instances de la classe. En plus, si chaque client doit contrôler le nombre d'instances de cette classe cela implique une responsabilité distribuée qui est une chose non souhaitable par ce qu'un client devrait être libre des détails du processus de création de toute classe.

Par conséquent, cette responsabilité devrait être assumée par la classe singleton elle-même, libérant ainsi les objets clients de tous ses détails. La figure VI.4.1 montre un exemple d'une implémentation du design pattern Singleton proposée par Hannemann and Kiczales [Hannemann 02]. Les éléments en ombré (bout de code étiqueté **A**) correspondent à la préoccupation du singleton. Nous remarquons le chevauchement du code de cette dernière avec le rôle principal de cette classe qui consiste en l'affichage des messages.

VI.5.2. Évaluation de la cohésion – ancienne approche

Selon les critères de cohésion étendue, décrits dans le chapitre II à la section II.2, on peut déduire le graphe non dirigé G_I correspondant à la classe *PrinterSingleton* qui est donné par le schéma de la figure VI.4.2:

```
public class PrinterSingleton
{
    protected static int objects = 0;
    protected static PrinterSingleton onlyInstance = null;
    protected int id;

    protected PrinterSingleton()
    {
        id = ++objects;
    }

    public static PrinterSingleton instance()
    {
        if (onlyInstance == null)        {
            onlyInstance = new PrinterSingleton();        }
        return onlyInstance;
    }

    public void print() {
            System.out.println("\tMy ID is "+id);
      }
}
```

A

Figure VI.4.1 : Implémentation du patron Singleton (la classe *PrinterSingleton)* [Hannemann 02].

Le nombre de méthodes est : N=3 ;

Le nombre de paires maximum est : Nmax = N *(N-1)/ 2 = 3;

Le nombre de paires de méthodes connectées est : NC=3 ;

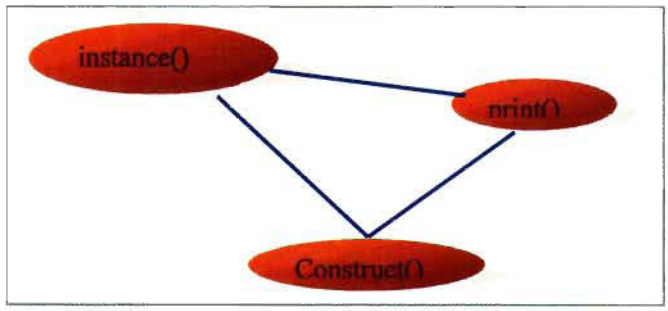

Figure VI.4.2 : Graphe non dirigé G_I correspondant la classe *PinterSingleton*.

Par conséquent, $DC_{IE} = NC/Nmax = 1$.
Cela indique que la classe *PrinterSingleton* a une cohésion parfaite, donc elle est d'une bonne conception.

VI.5.3. Évaluation de la cohésion – nouvelle approche

Appliquons la nouvelle approche sur la même classe *PinterSingleton.* Le graphe de la figure VI.4.3 représente son graphe de relations d'usage. La matrice entités-propriétés (tableau VI.4.1) est obtenue selon les spécifications du modèle défini à la section V.2.1.

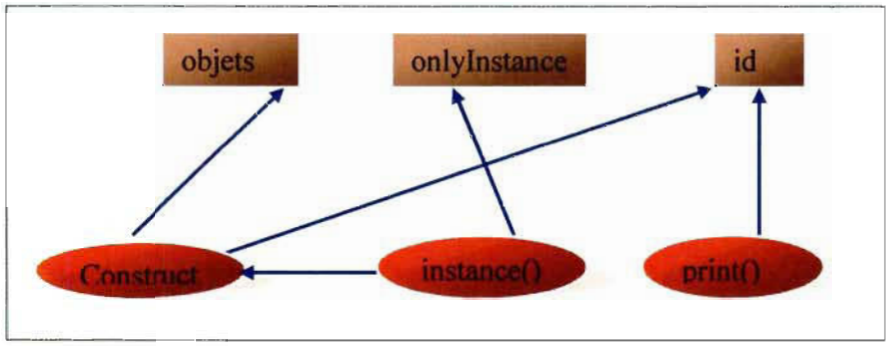

Figure VI.4.3 : Relation d'usage entre les membres de la classe *PinterSingleton.*

	propriétés					
objets	*objects*	*onlyInstance*	*id*	*construct()*	*instance()*	*print()*
construct()	1	0	1	0	0	0
instance ()	0	1	0	1	0	0
print()	0	0	1	0	0	0
objects	1	0	0	1	0	0
onlyInstance	0	1	0	0	1	0
id	0	0	1	1	0	1

Tableau VI.4.1 : Matrice Entités-propriétés de la classe *PinterSingleton.*

La matrice des proximités selon le coefficient de Jaccard, le tableau des statistiques des nœuds ainsi que l'arbre hiérarchique résultant de la procédure de classification hiérarchique sont donnés respectivement par les tableaux IV.4.2 et IV.4.3 et la figure VI.4.4.

	construct()	***instance ()***	***print()***	***objects***	***onlyInstance***	***id***
construct()	0,000	1,000	0,500	0,667	1,000	0,750
instance ()	1,000	0,000	1,000	0,667	0,667	0,750
print()	0,500	1,000	0,000	1,000	1,000	0,667
objects	0,667	0,667	1,000	0,000	1,000	0,750
onlyInstance	1,000	0,667	1,000	1,000	0,000	1,000
id	0,750	0,750	0,667	0,750	1,000	0,000

Tableau VI.4.2 : Matrice des proximités de la classe *PinterSingleton.*

Nœud	*Niveau*	*Poids*	*Objets*	*Fils gauche*	*Fils droit*
11	0,907	6	6	10	9
10	0,833	3	3	8	5
9	0,708	3	3	7	6
8	0,667	2	2	2	4
7	0,500	2	2	1	3

Tableau VI.4.3 : Statistiques des nœuds (la classe *PinterSingleton*).

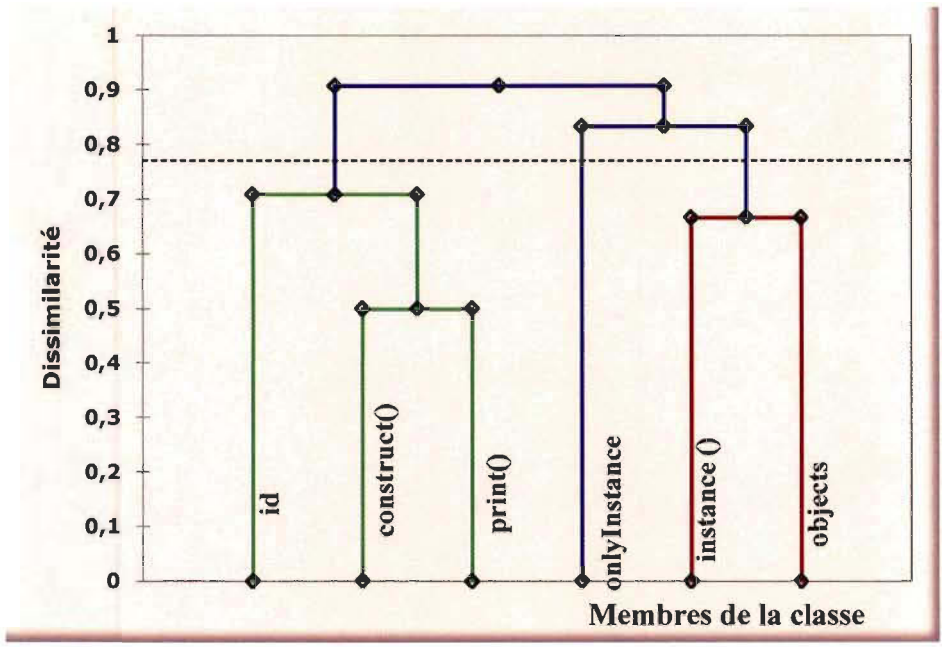

Figure VI.4.4 : Arbre hiérarchique correspondant à la classe *PinterSingleton*.

Selon le niveau de troncature automatique (XLSTAT) de l'arbre hiérarchique (figure VI.4.4), on obtient 2 clusters **par rapport la répartition des méthodes** comme suit :

1. **K1={*constructeur(), print()*}**
2. **K2={ *instance()* }**

Deux méthodes qui sont classées dans le même cluster sont (similaires) et donc reliées, d'où le graphe de connexions de la figure VI.4.5.

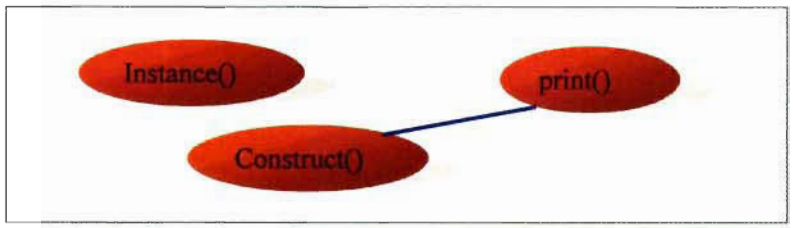

Figure VI.4.5 : Nouveau graphe de connexion de la classe *PinterSingleton* selon la nouvelle approche.

Le nombre de composantes connexes est : NCC=2.

Le nombre de méthodes du graphe est : N =3 ;
Le nombre total de paires de méthode du graphe est : Nmax= N*(N-1)/2 =3 ;
Le nombre de paires de méthodes connectées est : NC=1 ;

Par conséquent, COH_{CL}= NC/Nmax = 0.33.
Ce qui indique une faible cohésion pour la classe *PinterSingleton.*

VI.5.4. Discussion

Selon l'ancienne approche, la classe est considérée comme totalement cohésive puisque la valeur de la métrique DC_{IE} est égale à 1. Ce qui laisse penser que la classe ne présente aucune disparité dans le code; ce qui n'est pas le cas, puisque la classe contient du code transverse.

En appliquant la nouvelle approche, nous avons obtenu une valeur de 0.33 pour la métrique COH_{CL} qui est assez faible comme cohésion. Cela lève des doutes à l'égard de la classe analysée. Par ailleurs, comme la métrique NCC a une valeur de 2, alors cela

signifie que la classe comporte deux groupes disjoints, en l'occurrence **k1 et k2**. La classe est dotée de deux responsabilités différentes : Son rôle principal qui consiste à afficher les messages, et un rôle secondaire (de sécurité) pour s'assurer de l'instanciation d'un seul objet de cette classe. Les clusters k1 et k2 contiennent les méthodes qui correspondent exactement aux éléments de codes de ces deux préoccupations. La nouvelle approche a pu les isoler de façon appropriée.

Par conséquent, les valeurs de cohésion selon la nouvelle approche reflètent mieux la réalité de la classe quant à sa cohésion, et permettent de mieux déceler les signes qui justifieraient une éventuelle restructuration (classe candidate). Une meilleure conception de cette classe pourrait être réalisée, par exemple, via un refactoring aspect permettant de mieux encapsuler la préoccupation du singleton en extrayant ces parties dans un aspect illustré par la figure VI.4.6, en dehors de la classe de base (figure VI.4.7) comme suit :

Solution aspect

```
public aspect SingletonInstance {

  static Printer  Printer.onlyInstance;

      public static Printer Printer.instance() {

         if (onlyInstance ==null) onlyInstance= new Printer();
         return onlyInstance:

      }

}
```

Figure VI.4.6 : Aspect chargé de gérer la classe *singleton*.

Par conséquent, la classe *PrinterSingleton* sera réduite à sa fonctionnalité de base illustrée par le code de la figure VI.4.7.

```
public class Printer
{
    protected static int objects = 0;
    protected int id;

    public Printer()
    {
        id = ++objects;
    }

    public void print() {
            System.out.println("\tMy ID is "+id);
      }
}
```

Figure VI.4.7 : Code de la classe *Printer* après l'extraction du code enchevêtré.

VI.6. Bilan des résultats des expérimentations

Le tableau VI.5 donne un résumé sur les exemples analysés, les défauts de conception qu'ils contiennent et des résultats obtenus par les métriques de cohésion selon l'ancienne et la nouvelle approche. La courbe de la figure VI.5 donne une comparaison entre les valeurs des métriques DC_{IE} de l'ancienne approche et COH_{CL} et NCC de la nouvelle approche.

On constate que les valeurs de COH_{CL} sont toujours les plus faibles pour tous les exemples et reflètent mieux la réalité des classes analysées.

Exemple	Défaut de conception existant	Ancienne approche	Nouvelle approche	
		DC_{IE}	COH_{CL}	NCC
1 : Design pattern Observer	Classe modélisant deux responsabilités différentes.	1,000	0,400	2
2 : classe TangledStack	Classe contenant une préoccupation transverse	0,800	0,710	2
3 : D.P. chaîne de responsabilité	Présence de 2 responsabilités différentes	0.667	0,160	3
4 : D.P. singleton	Classe contenant une préoccupation transverse	1,000	0,333	2

Tableau VI.5 : Comparaison sommaire entre les deux approches de cohésion.

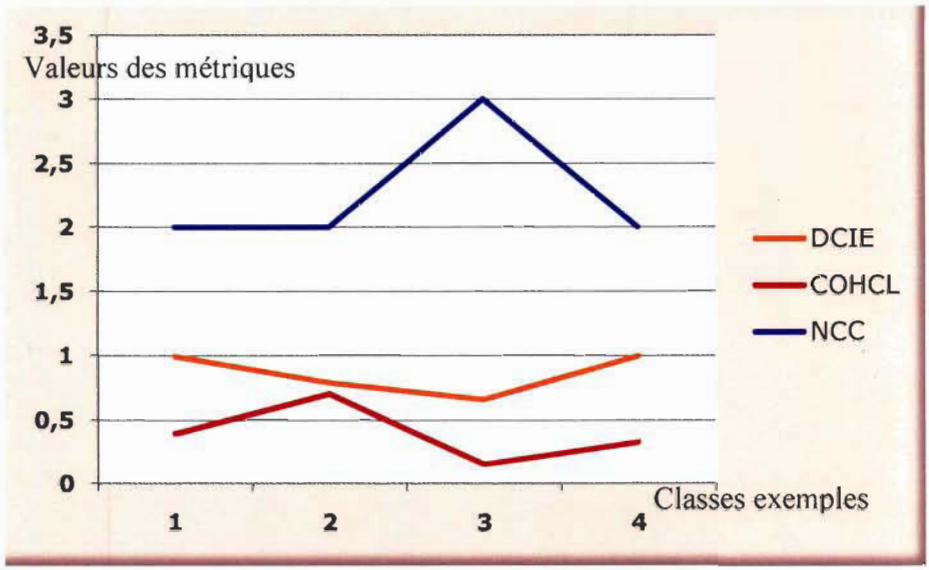

Figure VI.5 : Comparaison des tendances des métriques DC_{IE} et COH_{CL} et NCC.

Les résultats obtenus démontrent que la nouvelle approche reflète mieux la réalité des classes que l'ancienne approche. Elle a permis de capturer (refléter) plusieurs défauts de conception qui ont échappé à l'autre approche. Il s'agit d'anomalies liées à la mauvaise affectation de responsabilités aux classes et aussi à des problèmes liés à la présence de code correspondant à des préoccupations transverses.

CHAPITRE VII

CONCLUSION GÉNÉRALE

La nouvelle approche pour l'évaluation de la cohésion intègre les diverses interactions pouvant exister entre les membres d'une classe de manière plus consistante et significative contrairement aux anciennes approches structurelles. Elle permet, comme nous l'avons vu à travers divers exemples, de mieux différencier et classifier les membres d'une classe en groupes de méthodes reliées et cohésives. Cette approche a permis de mieux capturer les liens conceptuels existant entre les méthodes d'une classe. Cette capacité lui revient principalement grâce au potentiel de différenciation et de séparation des groupes cohésifs de la technique de classification. En effet, elle a permis d'exprimer efficacement comment les différentes méthodes d'une classe sont reliées ensemble au niveau conceptuel. Il nous a été possible de détecter si une classe est dotée de plus d'une fonctionnalité (responsabilité), ce qui est reflété par le nombre de groupes connexes dans cette classe.

Par ailleurs, nous pensons que notre approche pourrait aussi être utilisée pour supporter efficacement des opérations d'aspect mining et de refactoring des systèmes orienté-objet. En effet, quand les autres métriques (structurelles) ont échoué à déceler des signes de présence de disparité dans le code (due à la présence de code correspondant à des préoccupations transverses), notre métrique a pu le faire de façon efficace.

À travers les études de cas présentées et à la lumière des résultats obtenus, nous estimons que nos nouvelles métriques COH_{CL} et NCC sont bien adaptées pour refléter des problèmes de conception reliés à l'assignation de rôles disparates à une classe. Cette disparité peut éventuellement être due à la présence de préoccupations transverses.

RÉFÉRENCES BIBLIOGRAPHIQUES

[Aldenderfer 85] M. S. Aldenderfer et R. K. Blashfield. *Cluster Analysis*. Sage publications, los Angeles, 1985.

[Aman 02] H. Aman, K. Yamasaki, H. Yamada et MT. Noda. *A proposal of class cohesion metrics using sizes of cohesive parts*. Knowledge-based Software Engineering, T. Welzer et al. (Eds), pp. 102-107, IOS Press, September 2002.

[Badri 04] L. Badri et M. Badri. *A Proposal of a New Class Cohesion Criterion: An Empirical Study*. In Journal of Object Technology, vol. 3, no. 4, April 2004.

[Badri 08] Linda Badri, Mourad Badri et Alioune Badara Gueye. *Revisiting Class Cohesion: An empirical investigation on several systems*. In Journal of Object Technology, 2008.

[Bieman 95] J.M. Bieman et B.K. Kang. *Cohesion and reuse in an object-oriented system*. Proceedings of the Symposium on Software Reusability (SSR'95), Seattle, WA, pp. 259-262, April 1995.

[Briand 98] L.C. Briand, J. Daly et J. Wusr. *A unified framework for cohesion measurement in object-oriented systems*. Empirical Software Engineering, Vol.3, No.1, pp. 67-117, 1998.

[Candillier 06] L. Candillier. « *Contextualisation, Visualisation et Évaluation en Apprentissage non supervisé* ». Thèse de doctorat en informatique, Université Charles de Gaulle- Lille 3.Septembre 2006. Tirée du Site de l'auteur (Laurent Candillier): http://www.grappa.univ-lille3.fr/~candillier/clusters/ consulté le 072009.

[Chae 00] H. S. Chae, Y. R. Kwon et D H. Bae. *A cohesion measure for object-oriented classes*. Software Practice and Experience, No. 30, pp. 1405-1431, 2000.

[Chidamber 91] S.R. Chidamber et C.F. Kemerer. *Towards a Metrics Suite for Object-Oriented Design, Object-Oriented Programming Systems.* Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, Vol. 26, No. 10, pp. 197-211, October 1991.

[Chidamber 94] S.R. Chidamber et C.F. Kemerer. *A Metrics suite for object Oriented Design.* IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476-493, June 1994.

[Chidamber 98] S.R. Chidamber, David P. Darcy, et C.F. Kemerer. *Managerial use of metrics for object-oriented software: An exploratory analysis*. IEEE Transactions on Software Engineering, Vol. 24, No. 8, pp. 629-639, August 1998.

[Cooper 00] J. Cooper. *Java Design Patterns: A Tutorial.* Addison-Wesley, 2000.

[Czibula 06] Istvan G. Czibula et Gabriela Serban. *Improving Systems Design Using a Clustering Approach*. International Journal of Computer Science and Network Security (IJCSNS) 6, no. 12, 40{49, 2006.

[Czibula 07a] Grigoreta Sofia Cojocar, Gabriela Czibula et Istvan Gergely Czibula. *A comparative analysis of clustering algorithms in Aspect mining*. http://cs.ubbcluj.ro/~studia-i/2009-1/07-CojocarCzibula.pdf

[Czibula 07b] I.G. Czibula et G. Serban. *A hierarchical clustering algorithm for software systems design improvement*. KEPT 2007: Proceedings of the first International Conference on Knowledge Engineering: Principles and Techniques, August 2007, June 6, pp. 316{323.

[Czibula 07c] I. G. Czibula et G. Serban. *Hierarchical clustering for software systems restructuring*. INFOCOMP Journal of Computer Science, Brasil 6 (2007), no. 4, 43{51.

[Czibula 07d] G. Serban et I. G. Czibula. *Restructuring software systems using clustering*. ISCIS 2007: Proceedings of the 22nd International Symposium on Computer and Information Sciences, September 2007 November 7, pp. 33, IEE Explore.

[Czibula 08] Istvan Gergely, Czibula et Gabriela Czibula. *A partitional clustering algorithm for improving the structure of object-oriented software systems*. http://www.cs.ubbcluj.ro/~studia-i/2008-2/11-Czibula.pdf

[Czibula 09a] Gabriela Czibula, Grigoreta Sofia Cojocar, et Istvan Gergely Czibula. *A Partitional Clustering Algorithm for Crosscutting Concerns Identification*. In Proceedings of the International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS '09), pages 111{116, Cambridge, UK, February, 21-23 2009.

[Czibula 09b] Istvan Gergely Czibula, Gabriela Czibula, et Grigoreta Sofia Cojocar. *Hierarchical Clustering for Identifying Crosscutting Concerns in Object Oriented Software Systems*. In Proceedings of the 4th Balkan Conference in Informatics (BCI'09), Thessaloniki, Greece, September, 170-19 2009, submitted.

[DeMarco 82] T. DeMarco. *Controlling Software Projects*. Yourdon Press, New York, 1982.

[Diday 82] E. Diday, J. Lemaire, J. Pouget et F. Testu. «*Un algorithme de type nué dynamiques* ». pages 117-123. Dunond , 1982.

[Dudoit 02] S. Dudoit et J. Fridlyand. *A prediction-based resampling method for estimating the number of clusters in a dataset.* Genome Biology 3, research0036.1–0036.21, 2002.

[Etzkorn 03] L.H. Etzkorn, S.E. Gholston, J.L. Fortune, C.E. Stein, D. Utley, P.A. Farrington et G.W. Cox. *A comparison of cohesion metrics for object-oriented systems*. Information and Software Technology, 46 (10), pp. 677-687, 2004.

[Fielding 07] A.H. Fielding. *Cluster and classification techniques for the biosciences*. Cambridge University Press, Cambridge, 246 pp, 2007.

[Gelinas 05] J.F. Gélinas. « *mesure de la cohésion dans les systèmes orientés aspect* ». Master's thesis, Université du Québec à Trois-Rivières, Canada, Juillet 2005.

[Han 01] J. Han et M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[Hannemann 02] J. Hannemann et G. Kiczales. *Design pattern implementation in Java and AspectJ.* In Proceedings of the 17th Annual Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA), pages 161–173, 2002.

[Hartigan 75] J. Hartigan. *Clustering Algorithms*. New York: Wiley, 1975.

[He 04] L. He et H. Bai. *Aspect mining using clustering analysis*. Technical report, Jilin University, 2004.

[He 06] L. He et H. Bai. *Aspect Mining using Clustering and Association Rule Method.* International Journal of Computer Science and Network Security, 6(2A):247–251, February 2006.

[Henderson-Sellers 96] B. Henderson-Sellers. *Software Metrics*. Prentice Hall, 1996.

[Hutchens 85] D. Hutchens et V. Basili. *System Structure Analysis: Clustering with Data Bindings*. IEEE Transactions on Software Engineering, SE-I 1(8):749-757, 1985.

[Imran 04] Imran Baig. *Measuring Cohesion and Coupling of Object-Oriented Systems-Derivation and Mutual Study of Cohesion and Coupling*. Master's thesis, School of Engineering, Blekinge Institute of Technology, 2004.

[Kabaili 00] H. Kabaili, R.K. Keller, F. Lustman et G. Saint-Denis. *Class Cohesion Revisited: An Empirical Study on Industrial Systems.* Proceeding of the workshop on Quantitative Approaches Object-Oriented Software Engineering, France, June 2000.

[Kaufman 90] L. Kaufman et P. J. Rousseuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. New York: Wiley-Interscience, 1990.

[Krzanowski 88] W. J. Krzanowski et Y. T. Lai. *A criterion for determining the number of groups in a data set using sum of squares clustering*. Biometrics 44, 23–34, 1988.

[Lethbridge 99] N. Anquetil, C. Fourrier, et T. Lethbridge. *Experiments with hierarchical clustering algorithms as software remodularization methods*. In Proc. Working Conf. on Reverse Engineering, October 1999.

[Lethbridge 03] N. Anquetil et T. C. Lethbridge. *Comparative study of Clustering Algorithms and Abstract Representations for Software Remodularization*. IEE Proc. on Software, 150(3), pp.185-201, 2003.

[Li 93] W. Li et S. Henry. *Object oriented metrics that predict maintainability*. Journal of Systems and Software, Vol. 23, pp. 111-122, February 1993.

[Lung 04a] Xia Xu, Chung-Horng Lung, Marzia Zaman et Anand Srinivasan. "*Program Restructuring Through Classification Techniques*". IEEE International Workshop on, pp. 75-84, Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04), 2004.

[Lung 04b] C.-H. Lung et M. Zaman. *Using Clustering Technique to Restructure Programs*. Proc. of the Int'l Conf on Software Eng. Research and Practice, pp. 853-860, 2004.

[Lung 04c] C. Lung , M. Zaman et A. Nandi. *Applications of clustering Techniques to software partitioning, recovery and restructuring*. J. Syst. Softw. 73, 2, 227-244, Oct. 2004.

[Lung 98] C.-H. Lung. *Software architecture recovery and restructuring through clustering techniques*. Proc. of the 3rd Int'l Workshop on Sw Architecture, pp. 101-104, 1998.

[Maier 00] M.W. Maier et E. Rechtin. *The Art of Systems Architecting*. 2nd edition. CRC Press, 2000.

[Marcus 08] A. Marcus, D. Poshyvanyk et R. Ferenc. *Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems*. IEEE Trans. Softw. Eng. 34, 2, 287-300, Mar. 2008.

[Miguel 05] Miguel Jorge Tavares Monteiro. *Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts*. PhD thesis, Universidade do Minho, 2005. http://repositorium.sdum.uminho.pt/bitstream/1822/3331/1/Tese-%20Miguel%20Pessoa%20Monteiro.pdf

[Moldovan 06a] G. S. Moldovan et G. Serban. *Aspect Mining using a Vector Space Model Based Clustering Approach*. In Proceedings of Linking Aspect Technology and Evolution (LATE) Workshop, pages 36– 40, Bonn, Germany, AOSD'06, March, 20 2006.

[Moldovan 06b] Gabriela Serban et Grigoreta Sofia Moldovan. *A New k-means Based Clustering Algorithm in Aspect Mining*. In Proceedings of 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'06), pages 69{74, Timisoara, Romania, IEEE Computer Society, September, 26-29, 2006.

[Romesburg 84] Romesburg, C. H. *Cluster Analysis for Researchers*. Life Time Learning, 334pp, 1984.

[Serban 07a] Gabriela Serban et Grigoreta Sofia Cojocar. *A New Hierarchical Agglomerative Clustering Algorithm in Aspect Mining*. In Proceedings of 3rd Balkan Conference in Informatics (BCI'2007), pages 143{152, Sofia, Bulgaria, September, 27-29, 2007.

[Serban 08a] Istvan Gergely Czibula et Gabriela (Serban) Czibula. *Hierarchical Clustering Based Automatic Refactorings Detection.* http://www.wseas.us/e-library/transactions/electronics/2008/28-173.pdf

[Shepherd 05] D. Shepherd et L. Pollock. *Interfaces, Aspects, and Views.* In proceedings of Linking Aspect Technology and Evolution (LATE) Workshop, Chicago, USA, March 2005.

[Simon 01] F. Simon, F. Steinbruckner, et C. Lewerentz. *Metrics based refactoring.* in CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering. Washington, DC, USA: IEEE Computer Society, pp. 30–38,2001.

[Sneath 73] P. H. A. Sneath et R. R. Sokal. *Numerical Taxonomy: The Principles and Practice of Numerical Classification.* Series of books in biology, W. H. Freeman and Company, San Francisco, 1973.

[Snelting 00] G. Snelting. *Software reengineering based on concept analysis.* Proc. of the European Conf. on Software Maintenance and Reeng., pp. 1-8, 2000.

[Tibshirani 01] R. Tibshirani, G. Walther, et T. Hastie. *Estimating the number of data clusters via the gap statistic.* Journal of the Royal Statistical Society B 63, 411–423, 2001.

[Valdano 03] S. Valdano et J. Di Rienzo. *Discovering meaningful groups in hierarchical cluster analysis.* An extension to the multivariate case of a multiple comparison method based on cluster analysis, 2007. http://interstat.statjournals.net/YEAR/2007/abstracts/0704002.php

[Wiggerts 97] T. A. Wiggerts. *Using Clustering Algorithms in Legacy Systems Remodularization.* In Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97) (October 06 - 08, 1997). WCRE. IEEE Computer Society, Washington, DC, 33. 1997.

[Zhang 08] D. Zhang, Y. Guo, et X. Chen. *Automated aspect recommendation through clustering-based fan-in analysis.* In Proc. of the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE), pages 278–287, Sept. 2008.