Vers un Contrôle de la Qualité du Développement Orienté Objet

Linda BADRI & Mourad BADRI

Institut d'Informatique - Université de Annaba
BP. 12 El-Hadjar, 23000 Annaba, Algérie.

ABSTRACT

Since the last two decades, industrial systems requiring the development of complex softwares have appeared. These softwares are complex considering both their size and their structure. The quality management, during the development process of such softwares, and the estimation of the final product quality, have become a necessity. The Object-Oriented Development Approach is a new technology in the software development. The concepts introduced by this approach have been a large and interesting subject of studies by researchers and industrials. Measures of object-oriented softwares are necessary in order to realize and achieve expected benefits of object technology. Therefore, constant quality management requires high frequently measurement of qualitative metrics that promote a process of continual quality improvement, particulary on large-scale projects. Within the framework of a global Quality Assurance and Control Approach, this paper presents a metrics suite for Object-Oriented Software Development.

Key Words : Software Engineering, Software Quality, Quality Assurance and Control, Object-Oriented Software Development, Quality Metric.

1. INTRODUCTION

Les systèmes industriels actuels nécessitent le développement de logiciels de plus en plus complexes et exigent de leur part un haut niveau de qualité (Maintenabilité, Réutilisabilité, ...[4, 20]). Ces logiciels sont d'une complexité à la fois de taille et de structure. La maîtrise de la qualité de tels logiciels est devenue indispensable. Elle nécessite une gestion et un contrôle continus de la qualité de la production durant toutes les phases du processus de développement de ces logiciels.

Le développement orienté objet est une approche relativement récente dans le domaine de la production des logiciels. Les concepts introduits par cette approche ont apporté une réponse à de nombreux problèmes du Génie Logiciel et de l'Intelligence Artificielle. Ils permettent, en particulier, une meilleure maîtrise de la conception et de la complexité des logiciels, ainsi qu'une réutilisation plus facile de leurs composants.

Par ailleurs, les principaux travaux effectués par Boehm [3, 4] et McCall [18,19] sur la qualité du logiciel, sont toujours considérés comme fondamentaux dans le domaine. Depuis la réalisation de ces travaux, les efforts de recherche dans le domaine de la qualité du logiciel, se sont surtout axés sur le développement de nouvelles métriques [15]. En effet, l'importance des métriques dans le processus de développement des logiciels n'est plus à démonter. Cependant, la multitude de travaux qui ont proposé des métriques de qualité ont plutôt porté sur les approches traditionnelles et ne couvrent pas les spécificités de l'approche objet (classes, héritage, polymorphisme, ...). Par ailleurs, on constate que très peu de travaux proposant des métriques spécifiques au développement orienté objet [21, 10, 11], ainsi que des expériences de collecte de ce type de métriques [25] ont été réalisés à ce jour.

Les travaux effectués par Chidamber et Kemerer [10] ont été une première proposition intéressante dans le domaine des métriques

spécifiques au développement orienté objet. Dans leur papier [10], les auteurs proposent une famille de six métriques pour la méthode de conception orientée objet OOD de Booch [7]. Seulement, ces métriques ont plutôt porté sur les phases de conception détaillée et de codage. Or, le contrôle de la qualité du développement à l'aide d'un ensemble de métriques est, à notre avis, d'autant plus efficace lorsqu'il intervient, dans la mesure du possible, dès les premières phases du processus de développement. Il permet, ainsi, d'appréhender tôt la qualité du développement et d'apporter en conséquence des améliorations en préconisant des modifications précoces. C'est dans ce contexte, et dans le cadre du projet de recherche *QOOD* (Quality Assurance and Control Approaches for Object-Oriented Development), que s'est située notre recherche.

2. METHODOLOGIE DE NOTRE APPROCHE

L'un de nos principaux objectifs, dans le cadre de ce projet de recherche était, d'une part, de développer une approche globale pour le contrôle de la qualité du développement des applications orientées objet, supportée par un outil CASE et basée sur un ensemble assez large de métriques prenant en compte les spécificités de l'approche objet. D'autre part, et en premier lieu, on voulait que cet ensemble de métriques couvre d'abord et au mieux les premières phases du processus de développement et, en second lieu, soit le plus large possible de façon à ce qu'il couvre également tout le reste du cycle de développement de cette approche. Ceci nous permet d'assurer un contrôle précoce, continu et cohérent de la qualité de la production le long du cycle de développement, depuis les spécifications jusqu'à la phase de codage.

Par ailleurs, la meilleure approche pour un développement orienté objet commence par une perception objet du problème, à l'aide de méthodes d'analyse des exigences orientées objet. C'est dans ce contexte que nous avons décidé de nous intéresser à une méthode d'analyse et de conception orientée objet. C'est la méthode OOAD (Object-Oriented Analysis and Design) préconisée par Coad et Yourdon [12, 13] qui a été choisie. Elle est une synthèse de différentes disciplines. Elle est basée sur les meilleurs concepts de la modélisation de l'information, des langages de programmation par objets et des systèmes à base de connaissances. Ces concepts sont solidement basés sur : l'abstraction, l'encapsulation, l'héritage, l'association, la communication avec messages, ... etc. En particulier, elle respecte dans leurs grandes lignes les règles proposées par G. Booch [5, 6] et par B. Meyer [20].

Le choix des métriques, que nous proposons dans ce papier, à été influencé par les travaux de Chidamber et Kemerer [10]. Ces travaux nous ont parus être un point de départ très intéressant à notre recherche. Cependant, nous avons pensé qu'il était plus intéressant de proposer un ensemble de métriques plus large et, surtout plus raffiné, directement applicable dès les premières phases du processus de développement, afin de contrôler leur évolution, de les raffiner et de les élargir, le long du processus, à d'autres métriques spécifiques à chaque phase.

3. METRIQUES PROPOSEES ET PROCESSUS DE DEVELOPPEMENT

L'ensemble des métriques retenues, au stade actuel de notre recherche, est applicable dès les premières phases du processus de développement objet , sur une réelle méthode d'analyse et de conception orientée objet (OOA [12] et OOD [13]). Nous proposons, dans le cadre de notre approche, un outil (*STOOSA* - Supporting Tools for Object-Oriented Systems Analysis), supportant, dans sa version actuelle, les différentes étapes de la méthode OOAD préconisée par Coad et Yourdon et offrant un environnement d'évaluation des métriques retenues [1]. L'évaluation des métriques retenues, durant les premières phases du processus de développement, est réalisée à partir d'une analyse des différents modèles OOAD.

Par ailleurs, les métriques proposées ne sont pas spécifiques à la méthode OOAD de Coad et Yourdon. Elles présentent l'avantage d'être générales et peuvent être implémentées pour l'ensemble des autres méthodes de développement orienté objet telles que OMT [23], OOAD [8] et OOM [22]. A ce stade, elles peuvent, non seulement, nous fournir une

idée générale sur la qualité de la conception préliminaire, mais aussi nous donner, d'ores et déjà, une idée précise sur certains aspects importants de la future application (héritage, découpage en classes, communication entre classes, ... etc.).

Les métriques proposées sont valables pour tout le processus de développement objet. Elle seront, ainsi, évaluées au niveau de chaque phase, raffinées tout le long du processus de développement, nous permettant de suivre et, par conséquent, de pouvoir contrôler la qualité du développement d'une application à différents niveaux (figure 1). Lors de la phase de codage, les métriques retenues sont évaluées par analyse des différentes classes d'une application. Ce traitement est réalisé par différents analyseurs statiques. Dans la version actuelle du prototype de l'environnement supportant notre approche (figure 2), seules les applications développées en Smalltalk V et C++ sont prises en compte. Les résultats de l'analyse sont sauvegardés dans une base d'informations et exploités par différents outils.

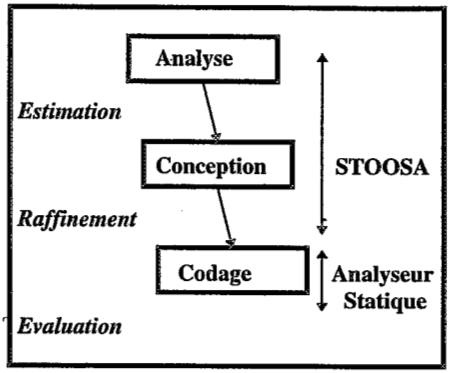

Figure 1 : Différents niveaux de contrôle.

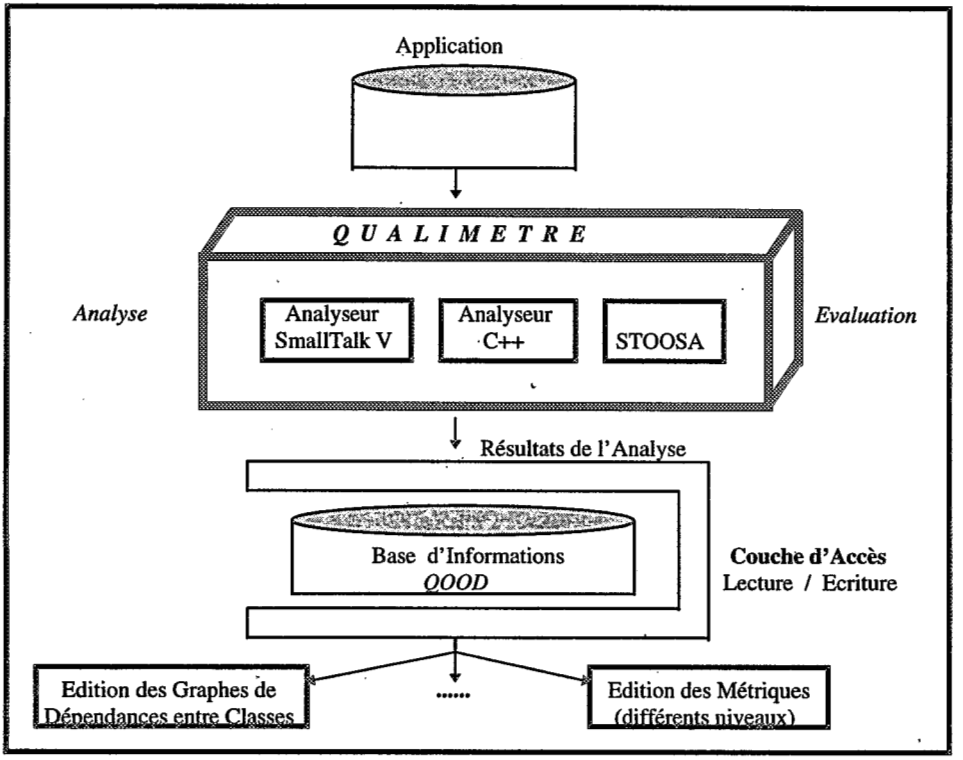

Figure 2 : Architecture générale de l'environnement.

4. METRIQUES PROPOSEES POUR LE DEVELOPPEMENT ORIENTE OBJET

4.1. Complexité d'une classe

On considère une classe C_i avec son ensemble d'attributs propres $A = \{A_1, A_2, ..., A_m\}$, son ensemble d'attributs hérités $AH = \{AH_1, AH_2, ..., AH_l\}$, son ensemble de méthodes propres $M = \{M_1, M_2, ..., M_n\}$, ainsi que son ensemble de méthodes héritées et non redéfinies $MH = \{MH_1, MH_2, ..., MH_k\}$. La complexité de la classe C_i peut être renseignée par plusieurs indicateurs :

Nombre d'Attributs Propres	**NAP**
Nombre d'Attributs Hérités	**NAH**
Nombre Total d'Attributs	**NTA**
Nombre de Méthodes Propres	**NMP**
Nombre de Méthodes Héritées	**NMH**
Nombre Total de Méthodes	**NTM**

Ces différentes métriques nous renseignent sur la richesse des propriétés de la classe. Le nombre de méthodes dans une classe, nous renseigne sur l'effort nécessaire à fournir pour le développement, le test unitaire et la mise au point de cette classe. Si le nombre d'attributs et le nombre de méthodes dans une classe sont trop élevé, cela peut limiter sa réutilisabilité, et peut avoir un impact sur ces descendants (complexité). Il faut peut être, dans ce cas, transformer la classe en une structure Généralisation-Spécialisation (héritage).

Complexité Textuelle d'une Méthode
Définition : La complexité textuelle [16] d'une méthode M_i est évaluée à partir de son texte source. A chaque méthode M_i on associe son indice de complexité textuelle ct_i.

Complexité Structurelle d'une Méthode
Définition : La complexité structurelle [17] d'une méthode M_i est évaluée à partir de son graphe de contrôle. A chaque méthode M_i on associe son indice de complexité structurelle cs_i.

Plus la méthode est complexe, plus l'effort nécessaire pour son test unitaire est important. Si la complexité de la méthode est importante, dès les premières phases du processus de développement (structurelle), il faut peut être penser à l'éclater en plusieurs méthodes (publique et privées). Supposons que ct_1, ct_2, ..., ct_n et cs_1, cs_2, ..., cs_n sont respectivement les indicateurs de complexité textuelle et les indicateurs de complexité structurelle des différentes méthodes d'une classe C_i.

Complexité Textuelle d'une Classe
Définition : La complexité textuelle d'une classe C_i peut être définie par : $CTC_{Ci} = \sum_i ct_i / n$, avec $i = 1,n$.

Complexité Structurelle d'une Classe
Définition : La complexité structurelle d'une classe C_i peut être définie par : $CSC_{Ci} = \sum_i cs_i / n$, avec $i = 1,n$.

Complexité Textuelle Etendue
Définition : La complexité textuelle étendue d'une classe C_i est calculée à partir des indices de complexité textuelle de l'ensemble de ses méthodes (propres et héritées).

Complexité Structurelle Etendue
Définition : La complexité structurelle étendue d'une classe C_i est calculée à partir des indices de complexité structurelle de l'ensemble de ses méthodes (propres et héritées).

La complexité d'une classe est estimée à partir de la complexité de ses méthodes. Plus ces dernières sont complexes, plus la complexité de la classe est importante. La complexité de la classe nous renseigne sur l'effort nécessaire pour son développement, son test unitaire et sa mise au point, ainsi que sa maintenance. Ces différents indicateurs nous renseignent également sur l'importance de l'impact que peut avoir cette classe sur ses descendants directs, en premier lieu, et sur l'ensemble de ses descendants d'une façon générale. Si la complexité de la classe est importante, à une étape donnée du processus de développement, il faut peut être penser à l'éclater en plusieurs classes (transformation en une structure Généralisation-Spécialisation).

4.2. Polymorphisme des méthodes

A chaque méthode M_i d'une classe est associé son identificateur (sélecteur de méthode) s_i. Le polymorphisme des méthodes est renseigné à l'aide des deux métriques suivantes :

Taux de Redéfinition d'une Méthode
Définition : TRM d'une méthode M_i est donné par le nombre de fois que la méthode est redéfinie le long d'un chemin d'héritage.

Taux de Polymorphisme d'une Méthode
Définition : TPM d'une méthode M_i est donné par le nombre de fois où son sélecteur s_i apparaît dans les différentes classes de l'application.

Ces deux métriques sont principalement utilisées lors des tests unitaires et d'intégrations des classes [2].

4.3. Héritage d'une classe

Il est intéressant, à plus d'un titre, de connaître la position d'une classe dans son graphe d'héritage. C'est ce que nous cherchons à travers les métriques suivantes :

Profondeur de la classe
Définition : La profondeur de la classe C_i, dans le graphe d'héritage, est donnée par la longueur du plus long chemin menant de la racine à cette classe.

Nombre d'ascendants directs d'une classe
Définition : DAC = nombre des super-classes immédiates de la classe C_i dans le graphe d'héritage.

Il est possible pour une classe d'hériter directement de plusieurs super-classes. L'héritage multiple est intéressant à plus d'un titre (accroître la modularité, mise en commun d'informations, ...). Cependant, le contenu et l'agencement des classes doivent être élaborés avec une grande attention. Lorsque le graphe d'héritage devient complexe, son exploitation devient très difficile. On peut donc, et suivant les cas, limiter le nombre de super-classes d'une classe à un seuil donné, ou du moins s'assurer que l'on ne peut pas faire mieux (revoir la hiérarchie).

Nombre d'ancêtres total d'une classe
Définition : C'est le nombre total des classes ancêtres de la classe C_i dans le graphe d'héritage.

Ces trois métriques, avec le graphe d'héritage intrinsèque ascendant de la classe (ensemble des classes ancêtres) (exemple : (1) pour la classe E dans la figure 3), nous donnent une idée précise sur l'importance (ampleur) de son héritage et, par conséquent, sur la richesse de ses propriétés (attributs et méthodes). D'autre part, cela nous permet de mettre en évidence de façon claire les classes fortement complexes figurant parmi les ancêtres d'une classe (impact).

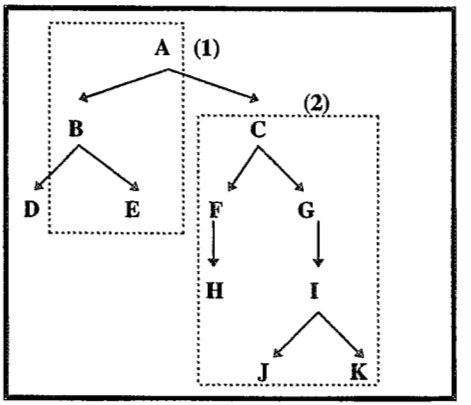

Figure 3: Graphe d'héritage.

Cela nous permet également d'apporter une aide dans l'orientation du processus de test de ce paquet de classes (unitaire). En effet, la classe en question ne peut être pleinement testée, lors de son test unitaire, que si toutes ses classes ancêtres ont été testées. De cette façon, on s'assure que toutes les propriétés héritées par une classe donnée ont été testées.

Nombre de sous-classes directes d'une classe
Définition : NCC = nombre de sous-classes immédiates de la classe C_i dans le graphe d'héritage.

Nombre de descendants total d'une classe
Définition : C'est le nombre total de classes du sous-graphe d'héritage dont la classe C_i est la racine.

Profondeur du graphe d'héritage intrinsèque descendant d'une classe
Définition : La profondeur du graphe d'héritage intrinsèque descendant d'une classe C_i est donnée par la longueur du plus long chemin menant de C_i aux feuilles.

Ces dernières métriques, avec le graphe d'héritage intrinsèque descendant d'une classe (exemple : (2) pour la classe C dans la figure 3), nous renseignent directement sur l'importance et l'étendue de l'impact que peut avoir cette classe sur l'ensemble de ses descendants. L'effort de test unitaire, nécessaire pour cette classe, doit être fonction de sa complexité ainsi que de l'importance de son impact sur ses éventuels descendants.

4.4. Ensemble de méthodes appelées dans une classe

On considère une classe C_i dans laquelle sont définies les méthodes M_1, M_2, ..., M_n. Supposons que $\{MA_i\}$ = ensemble des méthodes appelées par la méthode M_i (une méthode M_j peut apparaître plusieurs fois dans cet ensemble). Il existe alors n ensembles $\{MA_1\}$, $\{MA_2\}$, ..., $\{MA_n\}$. La cardinalité de l'ensemble MAC_i, ensemble de toutes les méthodes appelées dans la classe C_i, est un indicateur de l'effort requis pour le test et la mise au point de la classe. Il nous renseigne également sur la communication entre objets de l'application. En effet, la part des méthodes externes, dans l'ensemble total des méthodes appelées par la classe, nous renseigne sur l'ampleur de la communication de la classe avec les autres classes de l'application. Ceci est intéressant pour les tests d'intégration des classes. A chaque méthode M_i de la classe est associé un taux t_{Mi} donné par : t_{Mi} = Card(MA_i) / Card(MAC_i), indiquant la part de la méthode dans l'ensemble total des messages dans la classe.

4.5. Couplage entre classes

Le couplage à été défini, dans les approches traditionnelles, comme une mesure de l'interdépendance entre les différents modules d'une application [24]. Il a été introduit dans l'approche objet dans [9, 10, 11, 13]. Dans cette section, nous nous intéressons à différents types de couplage :

Couplage entre méthodes d'une classe

Soit une classe C_i avec son ensemble de méthodes M = $\{M_1, M_2, ..., M_n\}$.

Définition : Le couplage entre la méthode M_i et la méthode M_j d'une même classe, est défini par le nombre de fois que M_i appelle M_j dans son traitement.

Il permet de renseigner exactement l'interdépendance entre les différentes méthodes d'une même classe. Ceci peut être représenté à l'aide de la matrice de couplage intra-classe CIC : CIC[M_i,M_j]=n (nombre de fois que M_i appelle M_j). Elle permet d'indiquer le degré de dépendance des différentes méthodes de la classe entre elles. L'exploitation de cette matrice nous permet, d'une part, de déterminer en premier lieu l'ensemble minimal des méthodes sollicitées par toutes les autres, méthodes qu'il faut tester et intégrer en premier lors du test unitaire de la classe et, en second lieu, par réductions successives les méthodes qu'il faut tester par la suite. D'autre part, elle servira pour les critères d'arrêt des tests d'intégration entre les méthodes d'une même classe. On associe, à chaque couplage C(M_i,M_j), un taux de couverture de test TC_{ij}(M_i,M_j) : TC_{ij}(M_i,M_j) = E(M_i,M_j) / C(M_i,M_j) $\in$ [0,1], avec E(M_i,M_j) : nombre de messages exécutés, au moins une fois, entre M_i et M_j.

Couplage entre classes

On s'intéresse, dans un premier temps, au degré de liaison (interdépendance) d'une classe C_i à son environnement E_i : ensemble des classes qu'elle sollicite, et qui n'appartiennent pas à son graphe d'héritage (ancêtres).

Définition : Le couplage de la classe C_i à son environnement E_i est donné par le nombre total d'appels des méthodes définies dans les classes de E_i par les méthodes de la classe C_i.

On considère une classe C_i avec son ensemble de méthodes M=$\{M_1, M_2, ..., M_n\}$. Supposons que $\{MA_i\}$ = ensemble des méthodes appelées par la méthode M_i, où MA_i $\in$ C_j et C_j $\in$ E_i. Il existe n tels ensembles $\{MA_1\}$, $\{MA_2\}$, ..., $\{MA_n\}$. Le couplage entre C_i et E_i est donné par : C(C_i,E_i) = $\sum_j$ Card(MA_j), avec j = 1,n. La proportion de cet ensemble, par rapport à l'ensemble MAC_i, défini précédemment, nous renseigne sur sa dépendance fonctionnelle vis-à-vis de son environnement. Ceci peut être un bon indicateur sur la difficulté du test unitaire de la classe (effort de simulation de ce qui manque).

En effet, plus la classe est dépendante de l'extérieur, plus son test unitaire est difficile à réaliser. On s'intéresse, dans un second temps, au couplage de la classe C_i à toutes les classes C_j, appartenant à son environnement E_i, de façon individuelle.

Définition : Le couplage de la classe C_i à une classe C_j est donné par le nombre total d'appels des méthodes M_j de C_j par les méthodes M_i de C_i.

L'analyse de ce type de couplage nous donne une idée sur les détails de la dépendance fonctionnelle de la classe C_i vis-à-vis de son environnement. Cela permet aussi de mettre en lumière les classes auxquelles C_i est fortement couplée (classement des couplages), classes qui peuvent avoir une influence sur elle. Cela peut, également, nous aider à déterminer les classes à tester avant C_i, et à intégrer les classes auxquelles elle est fortement couplée en premier. A la limite, si le couplage entre C_i et une autre classe est trop fort, il est peut être préférable de les tester ensemble. On associe, à chaque couplage $C(C_i,C_j)$, un taux de couverture de test TC_{ij} : $TC_{ij}(C_i,C_j) = E(C_i,C_j) / C(C_i,C_j) \in [0,1]$, avec $E(C_i,C_j)$: nombre d'appels (messages) exécutés, au moins une fois, entre C_i et C_j. On associe, également, au couplage $C(C_i,E_i)$ un taux de couverture de test global qui peut rentrer dans les critères d'arrêt des tests d'intégration des classes : $TC(C_i,E_i) = E(C_i,E_i) / C(C_i,E_i) \in [0,1]$, avec $E(C_i,E_i)$: nombre d'appels exécutés, au moins une fois, entre C_i et E_i.

Dépendance fonctionnelle globale

L' analyse des différentes valeurs du couplage d'une classe C_i, vis-à-vis des autres classes de l'application, nous renseigne sur sa dépendance fonctionnelle globale dans l'application. Ceci est obtenu en calculant la part de chaque type de couplage dans l'ensemble total des méthodes que sollicite cette classe (MAC_i) (figure 4) :

IDH_i: indicateur de dépendance fonctionnelle de la classe vis-à-vis de ces ancêtres $\in [0,1]$,

IIC_i: indicateur de l'indépendance fonctionnelle de la classe (couplage inta-classe) $\in [0,1]$,

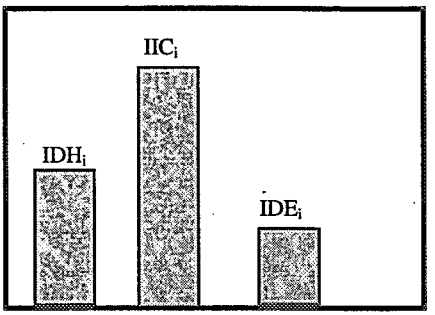

Figure 4 : Dépendance fonctionnelle.

IDE_i: indicateur de dépendance fonctionnelle de la classe par rapport au reste des classes de l'application $\in [0,1]$,
avec : $IDH_i + IIC_i + IDE_i = 1$.

Plus IIC_i d'une classe est élevé (proche de 1), plus son test unitaire est facile à réaliser (peu de simulations), et plus elle peut être facilement réutilisable.

4.6. Cohésion dans une classe

La cohésion a été définie, dans le cas des approches traditionnelles, comme une mesure du degré de liaison fonctionnelle des éléments d'un module [24]. Elle a été introduite dans l'approche objet dans [9, 10, 11, 13]. Dans cette section, nous nous intéressons à différents types de cohésion :

Cohésion par attributs

Soit une classe C_i avec l'ensemble de ses attributs propres $A=\{A_1, A_2, ..., A_m\}$ et l'ensemble de ses méthodes propres $M=\{M_1, M_2, ..., M_n\}$. Soit $\{A_i\}$ = l'ensemble des attributs utilisés par la méthode M_i. Il existe alors n ensembles $\{A_1\}, \{A_2\}, ..., \{A_n\}$. La cohésion à l'intérieur de la classe C_i peut être définie par la cohésion dans ses méthodes, qui peut être estimée par leur degré de similarité. Le degré de similarité des méthodes de la classe peut être donné par : $ds_1 = \{A_1\} \cap \{A_2\} \cap ... \cap \{A_n\}$. Ceci nous renseigne sur l'ensemble minimal des attributs A_i utilisés par l'ensemble des méthodes M_i de la classe. Plus cet ensemble est important, plus la cohésion intra-classe est forte. Le nombre d'ensembles disjoints formés par l'intersection des n ensembles indique la faible cohésion à l'intérieur de la classe. Le degré de cohésion

intra-classe est donné par : $dc_1 = Card(ds_1) / Card(A)$, $dc_1 \in [0,1]$. Si le degré de cohésion est faible (dc_1 tend vers 0), il faut peut être penser à éclater la classe. Quand le degré de cohésion est élevé (dc_1 tends vers 1), on parlera de cohésion forte.

Cohésion par méthodes privées

Soit une classe C_i avec l'ensemble de ses méthodes publiques $M=\{M_1, M_2, ..., M_n\}$ et l'ensemble de ses méthodes privées $R=\{R_1, R_2, ..., R_k\}$. Soit $\{R_i\}$ = l'ensemble des méthodes privées appelées par la méthode publique M_i. Il existe n tels ensembles $\{R_1\}$, $\{R_2\}$, ..., $\{R_n\}$. Le degré de similarité des méthodes de la classe C_i, dans ca cas, est donné par : $ds_2 = \{R_1\} \cap \{R_2\} \cap ... \cap \{R_n\}$. Le degré de cohésion intra-classe est alors donné par : $dc_2 = Card(ds_2) / Card(R)$, $dc_2 \in [0,1]$. Cela nous renseigne sur l'ensemble minimal des méthodes privées appelées par toutes les méthodes de la classe, ensemble qu'il faut, non seulement tester et intégrer en premier lors du test unitaire de la classe, mais aussi sur lequel il faut concentrer un effort de test important.

Cohésion globale dans la classe

L'approche adoptée pour l'estimation de la cohésion dans une classe par attributs (ou par méthodes privées) peut, dans certains cas, donner des valeurs nulles de l'indicateur dc_1 (ou dc_2). En effet, dans ces cas précis, nous pouvons dire qu'il n'y a pas de cohésion entre les différentes méthodes de la classe dans le sens où il n'y a pas un ensemble minimal commun d'attributs (ou de méthodes privées) utilisé par toutes les méthodes de la classe. Ceci ne doit pas exclure l'existence d'une possible bonne cohésion entre différents groupes de méthodes de la classe (par attributs, par méthodes privées ou par les deux). Sachant, par ailleurs, que les deux types de cohésion définis précédemment sont complémentaires, nous proposons alors, pour l'évaluation de la cohésion globale dans une classe, de les combiner. Cette combinaison est représentée à l'aide de la matrice de cohésion intra-classe COM, où $COM[M_i,Mj]$:

$$= \begin{cases} 1 \text{ si } \{A_i\} \cap \{A_j\} \neq \emptyset \text{ ou } \{R_i\} \cap \{R_j\} \neq \emptyset \\ 0 \text{ si } \{A_i\} \cap \{A_j\} = \emptyset \text{ et } \{R_i\} \cap \{R_j\} = \emptyset \end{cases}$$

Dans le cas où $COM[M_i,M_j] = 1$ (i=1,n et j=1,n), on peut dire qu'il y a une bonne cohésion entre les différentes méthodes de la classe, car elles sont corrélées entre elles soit par attributs, soit par méthodes privées, soit par les deux. Dans le cas contraire, on peut déterminer dans la matrice les différents blocs homogènes de 1. Ces derniers correspondent aux différents groupes $G_k = \{M_k / M_k \in C_i\}$ de méthodes corrélées entre elles, soit par attributs, soit par méthodes privées, soit par les deux. La taille S_{Gk} du bloc homogène G_k est donnée par : $S_{Gk} = Card(G_k)$. A chaque bloc homogène, on peut associer la proportion $P_{Gk} = S_{Gk} / Card(M)$. Il est également intéressant d'examiner le degré de similarité des différentes méthodes à l'intérieur d'un groupe homogène. Le degré de cohésion globale dans la classe C_i est donné par :

$$DGC_{Ci} = \sum_{i,j} COM[i,j] / ((n*(n-1))/2) \in [0,1]$$

avec : i = 1, n-1 et j = i+1, n. Le manque de cohésion dans la classe est alors donné par : $LCC_{Ci} = 1 - DGC_{Ci} \in [0,1]$.

4.7. Cohésion et Héritage

Soit une classe C_i avec son ensemble de méthodes $M = \{M_1, M_2, ..., M_n\}$. Soit $\{AH_i\}$= ensemble de tous les attributs hérités par la classe et utilisés par la méthode M_i. Il existe n tels ensembles $\{AH_1\}$, $\{AH_2\}$, ..., $\{AH_n\}$. La cohésion à l'intérieur de la classe C_i, par rapport aux attributs hérités, peut être définie par la cohésion de ses méthodes, qui peut être estimée par leur degré de similarité : $ds_3 = \{AH_1\} \cap \{AH_2\} \cap ... \cap \{Ah_n\}$. Le nombre d'ensembles disjoints, formés par l'intersection des n ensembles, est un indicateur de la faible cohésion intra-classe par rapport aux attributs hérités. Le degré de cohésion, dans ce cas, est donné par : $dc_3 = Card(ds_3) / NAH$, $dc_3 \in [0,1]$, avec NAH : nombre total d'attributs hérités par la classe. A ce niveau, il est intéressant d'avoir une vue globale, et voir l'allure de la courbe des indices de cohésion des classes le long d'un chemin d'héritage. Si l'indice de cohésion reste constant ou croit, le long du chemin, cela ne peut être qu'un bon indicateur de l'organisation de l'héritage.

4.8. Cohésion et encapsulation

Attributs définis et non utilisés

Soit une classe C_i avec l'ensemble de ses attributs propres $A=\{A_1, A_2, ..., A_m\}$ et l'ensemble de ses méthodes propres $M=\{M_1, M_2, ..., M_n\}$. La question que l'on se pose, dans ce cas, est de savoir s'il existe des attributs $A_k \in A$, tels qu'il n'existe aucune méthode $M_i \in M$ qui les utilise. Soit $\{M_i\}$ = l'ensemble des méthodes de la classe C_i qui utilisent l'attribut A_i. Il existe n tels ensembles $\{M_1\}$, $\{M_2\}$, ..., $\{M_m\}$. On s'intéresse à l'ensemble des attributs $ANU_{Ci} = \{A_k / A_k \in A$ et $\{M_k\} = \emptyset\}$. Cet ensemble est constitué par l'ensemble des attributs de la classe qui ne sont utilisés par aucune méthode de la classe.

Plusieurs cas de figure peuvent exister. Si $ANU_{Ci} \# \emptyset$, il existe alors un ou plusieurs attributs qui ne sont utilisés par aucune des méthodes de la classe C_i. Ceci conduit déjà à une mauvaise cohésion dans la classe et, dans certains cas, à des anomalies. En effet, si la classe en question est une feuille du graphe d'héritage de l'application, alors l'existence d'attributs non utilisés constitue une anomalie, sauf si le sous-graphe d'héritage, dont la classe C_i est racine, n'est pas encore développé. Si, dans ce dernier cas, et après avoir développé le sous-graphe d'héritage en question, certains des attributs de la classe ne sont toujours pas utilisés par les méthodes définies dans ses descendants, alors on peut parler d'anomalie dans le modèle. En effet, on se retrouve, dans ce cas, avec des attributs définis dans une classe et non utilisés, ni au niveau de cette classe ni même au niveau de ses descendants.

Dans le cas où les attributs définis et non utilisés dans la classe C_i sont utilisés par certaines méthodes définies dans ses descendants, on cherche à savoir, en réexaminant la hiérarchie des classes, si on ne peut pas définir les attributs en question ailleurs dans ses descendants, au meilleur emplacement dans la hiérarchie. Cette opération permet, non seulement, d'augmenter la cohésion dans la classe en question, mais aussi d'améliorer l'encapsulation en réduisant la visibilité des attributs. Elle permet aussi de faciliter le test unitaire des classes et leur réutilisation. Dans l'exemple de la figure 5, les deux attributs A_1 et A_2 sont définis au niveau de la classe C et utilisés au niveau des classes J et K. Le meilleur emplacement pour la définition de ces attributs serait la classe I, qui est le premier ancêtre commun aux deux classes J et K. Ce traitement est réalisé par un algorithme qui analyse la hiérarchie des classes (simple, multiple) [14], détermine pour chaque classe l'ensemble de ses attributs non utilisés, localise les différentes classes où sont utilisés les attributs en question, dans le graphe d'héritage intrinsèque descendant de la classe, et propose le meilleur emplacement pour les définir.

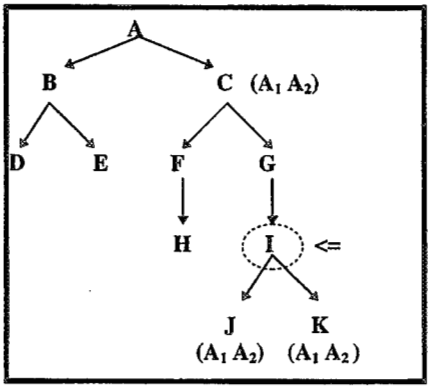

Figure 5 : Attributs non utilisés.

Méthodes définies dans une classe et n'utilisant aucun de ses attributs

Soit une classe C_i avec l'ensemble de ses attributs propres $A=\{A_1, A_2, ..., A_m\}$ et l'ensemble de ses méthodes propres $M=\{M_1, M_2, ..., M_n\}$. La question que l'on se pose, dans ce cas, est de savoir s'il existe des méthodes $M_k \in M$ qui n'utilisent aucun attribut $A_i \in A$, que ce soit de manière directe ou de manière indirecte par appel à une méthode privée.

Soit $\{A_i\}$ = l'ensemble des attributs de la classe qui sont utilisés par la méthode M_i. Il existe n tels ensembles $\{A_1\}$, $\{A_2\}$, ..., $\{A_n\}$. On s'intéresse à l'ensemble des méthodes $MNUA_{Ci} = \{M_k / M_k \in M$ et $\{A_k\} = \emptyset\}$. Cet ensemble est constitué par les méthodes qui n'utilisent aucun attribut de la classe. Une partie de cet ensemble peut correspondre aux méthodes héritées par la classe et redéfinies. On cherche à savoir, dans ce cas également, en réexaminant le graphe d'héritage intrinsèque ascendant de la classe, si on ne peut pas définir

les méthodes en question ailleurs dans ses ancêtres, au meilleur emplacement dans la hiérarchie. Nous avons développé, dans ce cas également, un algorithme qui analyse la hiérarchie des classes (simple, multiple), détermine pour chaque classe l'ensemble des méthodes qui n'utilisent aucun attribut local, et propose, en fonction de la hiérarchie, le meilleur emplacement pour les définir dans les ancêtres de la classe en question.

4.9. Complexité du graphe d'héritage

A ce niveau, on s'intéresse à l'ensemble des graphes d'héritage dans une application. Les métriques suivantes seront évaluées pour chaque graphe d'héritage.

Profondeur du graphe d'héritage

Définition : La profondeur d'un graphe d'héritage G_i, est définie par la longueur du chemin le plus long existant dans le graphe, parmi les différents chemins menant de la racine aux feuilles. Autrement dit, elle est égale à la plus grande des profondeurs des feuilles : $PGH = Max\ (P_i)$, avec : $i = 1,n$, où n : nombre de feuilles dans le graphe d'héritage, et P_i : longueur du chemin le plus long menant de la racine à la feuille i.

Nombre Moyen de Fils

Définition : Pour chaque niveau du graphe d'héritage, on évalue le nombre moyen de fils (NMF).

Nombre total de classes

Définition : NTC = nombre total de classes dans le graphe d'héritage.

Ces métriques nous donnent, non seulement une idée sur la profondeur du graphe d'héritage, mais aussi sur sa largeur. Elles peuvent servir comme indicateur du degré de spécialisation dans le graphe d'héritage. NMF, en général, tend à décroître de la racine vers les feuilles de la hiérarchie.

Dépendance Hiérarchique entre Classes

Définition : On considère un graphe d'héritage G_i, avec $C = \{C_1, C_2, ..., C_n\}$ l'ensemble de ses classes et $R = \{R1, R2, ..., Rm\}$ l'ensemble des relations (dépendances hiérarchiques) entres ces classes. Une relation R_i lie une classe C_j à celle dont elle hérite C_k (super-classe). Une classe peut hériter de plusieurs classes. La cardinalité de l'ensemble C étant n et celle de l'ensemble R étant m.

Plus les classes ont des super-classes directes dans le graphe, plus ce dernier devient complexe. La multiplicité de l'héritage ou la densité du graphe (dépendance hiérarchique) peut être renseignée à l'aide de l'indice suivant : $DHC = n / (m+1)$, $DHC \in [0,1]$.

Dans le cas d'un graphe d'héritage simple DHC = 1. Plus DHC tend vers zéro, plus il y a des dépendances hiérarchiques R_i dans le graphe d'héritage (héritage multiple complexe). Plus le graphe d'héritage est complexe, plus il est difficile de l'utiliser. Ceci peut également limiter la réutilisation de ses classes. Dans le cas d'une forte complexité, il faut peut être revoir la conception et proposer une autre hiérarchie.

5. CONCLUSIONS

Cette recherche nous a permis de développer un ensemble important de métriques pour la qualité du développement des applications orientées objet. Les métriques, que nous avons proposées dans ce papier, prennent en compte toutes les spécificités de l'approche objet. Elles présentent l'avantage d'être applicables dès les premières phases du processus de développement. Les mesures obtenues sont raffinées tout le long du processus de développement, au niveau de chaque phase (conception détaillée et codage), en fonction des détails qu'elle apporte (complexité des méthodes, messages, ...). De cette façon, elles nous permettent de suivre et contrôler l'évolution de la qualité du développement des applications durant tout le processus (analyse, conception, codage), de la gérer, donc de pouvoir la maîtriser.

Cette étude ne s'est pas arrêtée au stade théorique. Les métriques proposées ont été implémentées pour une réelle méthode d'analyse et de conception orientée objet. Elles présentent également l'avantage d'être générales et peuvent être appliquées aux autres méthodes de développement orientées objet. Par ailleurs, il est important de remarquer que les mesures obtenues par notre

approche ne sont pas absolues. Elles fournissent, cependant, des guides quantitatifs et relatifs qui permettent :

- d'appréhender tôt la qualité du développement,
- d'apporter une aide dans la conception,
- de détecter les parties les plus complexes,
- de quantifier les dépendances entre classes,
- d'orienter, en partie, le processus de test,
- et enfin, d'assurer un contrôle continue de la qualité du développement.

Grâce à l'environnement développé supportant notre approche nous travaillons, actuellement, sur une expérience de collecte de mesures. L'objectif de cette collecte et, d'une part, l'expérimentation des métriques retenues sur des projets réels et, d'autre part, une étude des différentes corrélations qui peuvent exister entre ces métriques et certains facteurs de qualité (testabilité, réutilisabilité, ...).

REFERENCES

[1]. L. BADRI, M. BADRI and S. FERDENACHE, « Towards Quality Control Metrics for Object-Oriented Systems Analysis », in Proceedings of the sixteenth International Conference TOOLS Europe'95 (Technology of Object-Oriented Languages ans Systems), Versailles, France, Prentice Hall International, March 1995.

[2]. M. BADRI, L. BADRI et S. LAYACHI, « Vers une Stratégie de Tests Unitaires et d'Intégration des Classes dans les Applications Orientées Objet », Revue Génie Logiciel & Systèmes Experts, Paris, à paraître.

[3]. B.W. BOEHM, « Characteristics of Software Quality » , North Holland, 1975.

[4]. B.W. BOEHM, J.R. BROWN and M. LIPOW, « Quantitative Evaluation of Software Quality », in IEEE 2nd International Conference on Software Engineering, 1976.

[5]. G. BOOCH, « Software Engineering with Ada », 2nd edition, The Benjamin/Cummings Publishing Company, Inc., 1986.

[6]. G. BOOCH, « Object-Oriented Development », IEEE Transactions on Software Engineering, SE-12, February 1986.

[7]. G. BOOCH, « Object-Oriented Design with Applications », Redwood City, CA : Benjamin / Cummings, 1991.

[8]. G. BOOCH, « Object-Oriented Analysis and Design with Applications », Benjamin / Cummings, 2nd edition, 1994.

[9]. T.A. BUDD, « An Introduction to Object-Oriented Programming », Addison-Wesley, 1991.

[10]. S.R. CHIDAMDER and C.F. KEMERER, « Towards a Metrics Suite for Object-Oriented Design », in Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, Vol. 26, N° 10, pp. 197-211, October 1991.

[11]. S.R. CHIDAMDER and C.F. KEMERER, « A Metrics Suite for Object-Oriented Design », IEEE Transactions on Software Engineering, Vol. 20, N° 6, pp. 476-493, June 1994.

[12]. P. COAD and E. YOURDON, « Object-Oriented Analysis », Yourdon Press Computing series, Prentice Hall, 1990.

[13]. P. COAD and E. YOURDON, « Object-Oriented Design », Yourdon Press Computing series, Prentice Hall, 1991.

[14]. S. FERDENACHE, « Indicateurs de Qualité pour le Développement Orienté Objet », Thèse de Magister, Université de Annaba, Institut d'Informatique, à paraître.

[15]. T. FORSE, « Qualimétrie des Systèmes Complexes : Mesure de la qualité du logiciel », Les Editions d'Organisation, 1989.

[16]. M.H. HALSTEAD, « Element of Software Science », North Holland Inc., 1977.

[17]. T.J. MAC CABE, « A complexity measure », IEEE Transactions on Software Engineering, SE-2, N° 4, 1976.

[18]. J.A. McCALL, « Factors in Software Quality », General Electric Compagny, 1977.

[19]. J.A. McCALL, « Software Quality Metrics », General Electric Compagny, 1979.

[20]. B. MEYER, « Object-Oriented Software Construction », Prentice Hall International, 1988.

[21]. D.R. MOREAU and W.D. DOMINICK, « Object-Oriented Graphical Information System : Research plan and evaluation metrics », in the Journal Systems and Software, Vol. 10, 1989.

[22]. A. ROCHFELD, « Modèle externe de données, modèle externe objet », Actes des 4èmes Journées Internationales sur le Génie Logiciel et ses Applications, Toulouse, 1991.

[23]. J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY and W. LORENSEN, « Object-Oriented Modeling and Design », Prentice Hall, 1991.
[24]. W. STEVENS, G. MYERS and L. CONSTANTINE, « Structured Design », IBM Systems Journal, Vol. 13, N° 2, 1974.
[25]. B. STIGLIC, M. HERICKO and I. ROZMAN, « How to Evaluate Object-Oriented Sostware Development ? », ACM SIGPLAN Notices, Vol. 30, N° 5, May, 1995.