Détection d'anomalies utilisant un langage de description de règle de qualité

El Hachemi Alikacem* — Houari A. Sahraoui**

* *CRIM : Centre de recherche informatique de Montréal*
550, rue Sherbrooke Ouest, bureau 100
Montréal, Qc, Canada H3A 1B9
el-hachemi.alikacem@crim.ca

** *Département d'informatique et de recherche opérationnelle*
Université de Montréal
CP. 6128, succ Centre-ville
Montréal, Qc, Canada H3C 3J7
sahraouh@iro.umontreal.ca

RÉSUMÉ. *De nombreuses règles de qualité ont été proposées dans la littérature. Celles-ci correspondent aux « bonnes pratiques » pour concevoir et implémenter des programmes à objets, ainsi que pour éviter d'introduire des défauts dans ces programmes. Malheureusement, la nature de ces règles fait qu'il n'existe pas d'outils qui puissent les exploiter complètement pour améliorer la qualité dans le développement par les objets. Dans cet article, nous présentons une approche pour la détection de violation des règles de qualité dans les programmes à objets. Nous proposons de modéliser ces règles par des systèmes à base de règles floues agissant sur le code source.*

ABSTRACT. *Numerous quality rules have been proposed in the literature. These rules define the "best practices" for designing and to implementing flaw-free object-oriented program. Unfortunately, due to the particular nature of these rules, there are no existent tools that can really take advantage of them to improve the quality in object-oriented development. In this paper, we propose an approach for detecting violations of quality rules in object-oriented programs. We suggest modeling these rules using fuzzy rule-based systems and apply them on source code.*

MOTS-CLÉS : *Évaluation de la qualité, analyse statique, métriques du logiciel*

KEYWORDS: *Quality assessment, static analysis, software metrics.*

1. Introduction

La technologie des objets est largement adoptée aujourd'hui dans le développement des logiciels. De nombreuses méthodes, techniques et langages encadrent efficacement ce développement. Parallèlement, les nombreuses années de pratique dans le développement par les objets ont permis d'établir des heuristiques de conception pour la production des logiciels de qualité en évitant les défauts de conception et d'implémentation.

De nombreuses règles de qualité matérialisant les heuristiques et les connaissances acquises ont été proposées dans la littérature. Cependant, notre expérience et nos études sur l'état de la pratique montrent que le développement par les objets ne tire pas suffisamment profit de ces acquis. Ainsi, quand elles ont lieu, les inspections sont effectuées manuellement par des experts qui appliquent implicitement ou explicitement des règles de qualité. Toute automatisation se heurte à des problèmes de contexte qui rend l'interprétation de ces règles difficile. Par exemple, si une règle préconise d'éviter les grandes classes, il est très difficile de définir de manière objective la limite au-delà de laquelle une classe sera considérée comme grande.

Dans cet article, nous proposons une solution pour la détection des non-conformités avec les règles de qualité dans le code source. Il s'agit d'un environnement qui offre aux utilisateurs un langage de description des règles de qualité, ainsi que des métriques nécessaires à la modélisation de ces règles. Pour cela, nous utilisons la logique floue pour exprimer, entre autres, les valeurs seuils des conditions des règles. Les règles mettent en jeux à la fois des informations quantitatives et de l'information structurelle, telles que les relations d'héritage ou d'associations entre classes, présentes dans le code source.

La suite de cet article est organisée comme suit. Nous présentons un aperçu de notre approche dans la section 2. La section 3 décrit le méta-modèle de représentation du code source. La technique de collecte des métriques qui serviront à la détection est introduite dans la section 4. La section 5 est dédiée à la modélisation des règles de détection. Pour situer notre approche par rapport aux autres travaux, un bref état de l'art est présenté dans la section 6. Finalement, une conclusion est donnée dans la section 7.

2. Aperçu de l'approche

Pour être efficace, un environnement de détection de non conformités avec les règles de qualité doit être flexible. En effet, en raison de la diversité des règles, celles-ci ne peuvent être implémentées « en dur ». Il faut au contraire offrir aux futurs utilisateurs de notre environnement un moyen flexible leur permettant de décrire et de modifier les règles qu'ils désirent mettre en œuvre.

Pour définir un environnement flexible, nous proposons une architecture à trois constituants comme le montre la figure 1 : (1) représentation du code source, (2) collecte des métriques et (3) détection de non conformités.

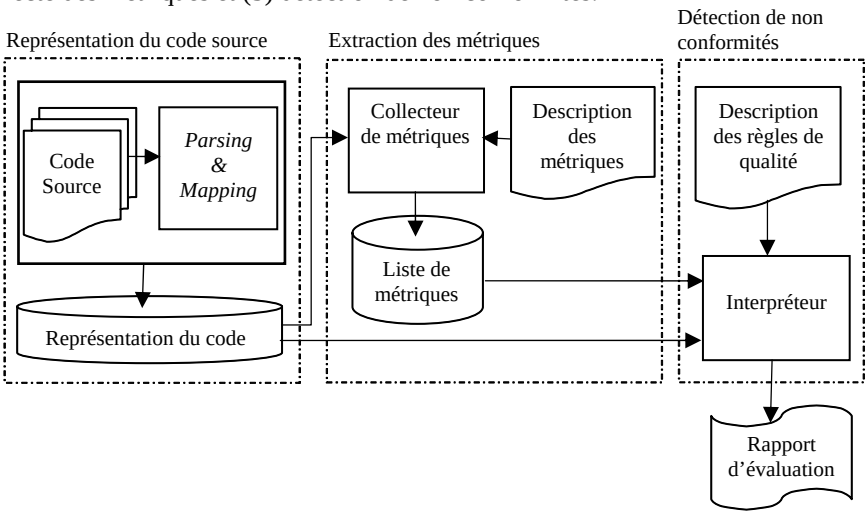

Figure 1. *Architecture du système d'évaluation de la qualité*

La représentation du code a pour objectif d'extraire à partir du code source les informations utiles à l'évaluation des règles de qualité pour les représenter selon un méta-modèle défini. Cette représentation permet une manipulation aisée de l'information d'une part et une analyse indépendante du langage de programmation d'autre part.

Une composante importante dans notre environnement est l'extraction des métriques. Cette extraction est nécessaire car de nombreuses règles de qualité mettent en jeu des notions quantitatives.

Le dernier constituant de notre environnement est le module d'évaluation des règles de qualité (appelé interpréteur). Cette évaluation met en jeu des aspects quantitatifs (métriques) et des aspects structurels contenus dans la représentation du code.

3. Représentation du code source

Afin de collecter l'information quantitative et structurelle nécessaire à l'évaluation des règles de qualité, le code source est représenté sous une forme aisément manipulable. À cet effet, nous avons défini un méta-modèle destiné à la

représentation des programmes analysés (voir une vue partielle du méta-modèle dans la figure 2).

Pour permettre une meilleure applicabilité de notre solution, notre méta-modèle est destiné à représenter des programmes à objets typés écrits dans différents langages. Ainsi, pour adapter l'environnement à un langage particulier, il suffit d'écrire un module spécifique à ce langage. Ce module permet de générer une représentation conforme à notre méta-modèle. Les autres étapes de l'analyse (collecte de métriques et évaluation des règles) demeurent ainsi indépendantes du langage de programmation.

La réalisation de cette indépendance n'est pas chose aisée. En effet, même si les langages proposent des concepts syntaxiquement équivalents, il demeure que la sémantique associée à ces concepts peut varier considérablement d'un langage à un autre (voir des exemples de ces variations dans (Beugnard, 2005)). Sans prétendre réaliser de manière complète cette indépendance et en nous restreignant aux informations nécessaires à notre analyse, nous proposons une solution qui comporte 3 volets correspondant chacun à une catégorie de concepts. Ainsi, pour un langage donné, nous pouvons classifier les concepts dans 3 catégories : (1) les concepts de base dont la syntaxe et la sémantique sont équivalentes à celles de la majorité des langages, (2) les concepts dont la syntaxe est équivalente mais dont la sémantique varie, et enfin (3) les concepts qui sont spécifiques au langage considéré.

Pour représenter les concepts de la première catégorie, nous avons défini la notion d'*Entity* qui peut se spécialiser selon les langages en concepts spécifiques tels que *ClassDef, InterfaceDef, Structure, Union et Template*. À ces notions, s'ajoutent celle de *Feature* qui peut elle aussi se spécialiser en *Attribute*, *Method*, ou *Constructor*, ainsi que celles de *Parameter* et de *Variable*. Finalement, cette liste est complétée par certaines catégories d'*Instruction* et d'*Expression*. Pour ces concepts, la transformation est directe.

Les différentes relations entre les concepts précédents constituent pour la majorité les constituants de la deuxième catégorie, i.e. les concepts ayant des sémantiques différentes. Par exemple la relation d'héritage entre deux classes n'a pas la même sémantique dans Java et dans C++. Pour contourner ce problème, nous avons décidé d'interpréter ces relations lors de la génération de la représentation. Dans le cas de l'héritage, les méthodes et les attributs sont dupliqués dans les sous-classes selon la sémantique propre à chaque langage. Ainsi, au moment du calcul des métriques et de l'évaluation des règles, nous n'avons pas à savoir si on manipule l'héritage dans Java ou dans C++. Les autres concepts qui sont interprétés sont le polymorphisme (détermination des méthodes redéfinies), les relations entre méthodes et attributs (quelle méthode manipule quel attribut), l'invocation (quelle méthode invoque quelle méthode), et les portées. Il est à noter que l'interprétation est réalisée dans les limites de ce que permet l'analyse statique. Par ailleurs, le fait que les relations susmentionnées soient interprétées ne les empêche pas d'être mentionnées explicitement dans la représentation. En effet, pour l'évaluation de

certaines règles, la mention explicite des relations est nécessaire. C'est le cas par exemple des règles qui évaluent les arbres d'héritage.

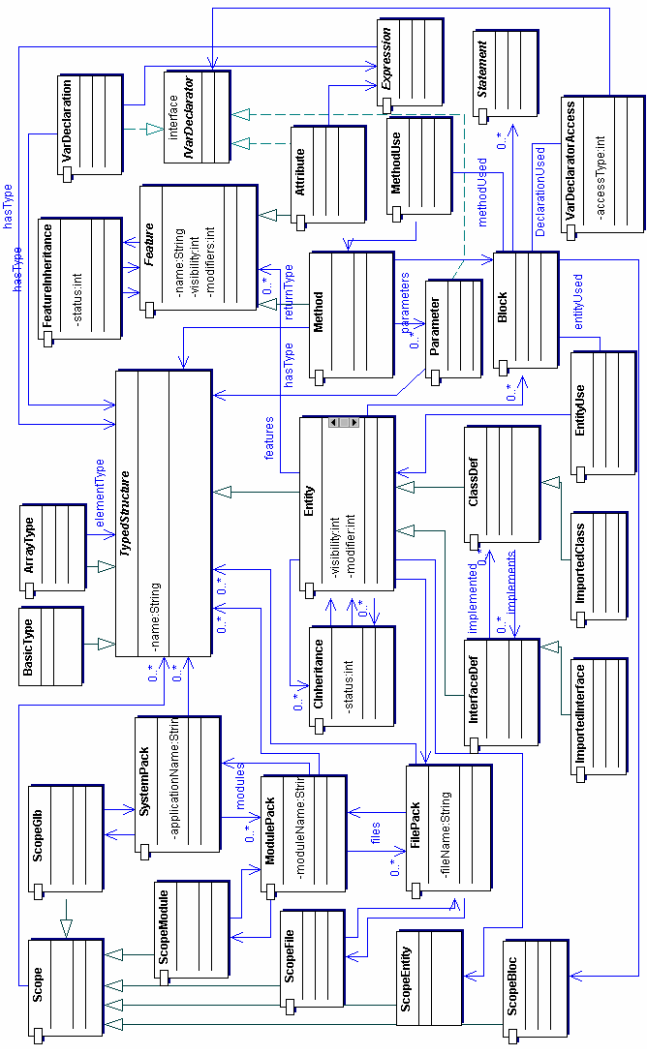

Figure 2. *Modèle de représentation*

L'interprétation de la sémantique de certaines constructions syntaxiques ne modifie en rien le programme analysé. Elle permet uniquement d'enrichir la représentation tout en levant les ambiguïtés inhérentes aux variations sémantiques des langages.

4. Collecte des métriques

Comme nous l'avons mentionné précédemment, une des exigences de l'environnement que nous voulons développer est la flexibilité. À tout moment, il est possible d'ajouter ou de modifier des règles de qualité. Ces deux opérations exigent parfois d'ajouter de nouvelles métriques. La majorité des outils de collectes de métriques existants imposent de *coder en dur* ces dernières. Nous avons alors, choisi de développer une approche nouvelle permettant de réduire l'effort quand vient le temps d'ajouter une nouvelle métrique.

Les métriques sont calculées en utilisant l'information présente dans la représentation du code source. Le calcul d'une métrique implique les notions suivantes :

- **Primitives :** Ce sont des ensembles de base extraits à partir de la représentation. Parmi ces ensembles, on trouve, par exemple, l'ensemble des classes du programme (*classes()*), les ensembles des méthodes et des attributs d'une classe (respectivement *methods(c*) et *attributes(c)*), les ensembles des enfants et des parents d'une classe (respectivement *parents(c)* et *children(c)*), l'ensemble des paramètres d'une méthode (*parameter(m)*) etc. Aux éléments de ces ensembles s'appliquent les propriétés (i.e. les attributs et les libellés des associations) présentées dans le modèle (figure 2), par exemple, $m \rightarrow$ *files* retourne l'ensemble des fichiers du module *m*, sachant que *files* est le libellé de l'association dans le modèle. De même, l'opération *c.visibility* retourne la visibilité de la classe *c* telle que définie dans le code source.
- **Opérateurs :** Ce sont des opérateurs arithmétiques, des opérateurs de comparaison, des opérateurs ensemblistes et des fonctions telles *Min, Max, etc*.
- **Itérateur :** C'est une structure de la forme

forAll *(x :inputSet ; condition ;* ***SET*** *operator expression)*

Où *inputSet* est l'ensemble parcouru, *condition* est la condition qui est évaluée sur chacun des éléments de l'ensemble parcouru, *operator* un opérateur d'affectation et *expression,* l'expression dont le résultat de l'évaluation est affecté à ***SET***.

Par exemple, le calcul de la métrique CLS (nombre de classe dans le système) est exprimé en utilisant la primitive *classes()* et l'opérateur de cardinalité | | comme suit

$$CLS = |classe()|$$

Comme autre exemple, nous montrons le cas NIC (nombre de classes indépendantes). Une classe est dite indépendante si elle ne possède pas de parents ni d'enfants. Le calcul de NIC s'exprime comme suit

$$NIC = \left| forAll(x : classe(); \ \left|parent(x)\right| == 0 \ \ \&\& \ \ \left|children(x)\right| == 0; \ \ SET+ = x) \right|$$

Une propriété intéressante de notre approche est qu'une métrique peut être utilisée dans le calcul d'une autre métrique. Par exemple, si l'on veut calculer AID (la profondeur moyenne des classes dans l'arbre d'héritage, on utiliserait les métriques DIT(c) (profondeur de la classe c dans l'arbre d'héritage) et CLS comme suit

$$AID = \frac{sum(\ forAll(x : classe(); \ ; \ \ SET+ = DIT(x)))}{CLS}$$

Comme les primitives, les opérateurs et l'itérateur sont préalablement implémentés, la valeur de la métrique est obtenue par l'évaluation de son expression. Il n'est nécessaire d'écrire du code que si le calcul d'une métrique exige la définition d'une nouvelle primitive. Ceci, cependant, n'arrive que très rarement compte tenu du grand ensemble des primitives déjà implémentées.

En pratique, les expressions des métriques sont interprétées par le collecteur de métriques (voir figure 1). L'interprétation utilise les différents opérateurs et primitives préalablement implémentés.

5. Évaluation des règles de qualité

Comme nous l'avons mentionné dans la section 2, le module d'interprétation permet de détecter les anomalies de conception/programmation en évaluant les descriptions des règles de qualité écrites dans un langage déclaratif conçu pour cet effet. L'évaluation se fait à partir des métriques et des informations structurelles contenues dans la représentation du code source.

L'effort de transcription des règles dans notre langage de description varie selon le degré d'abstraction de ces règles. Pour certaines, la transcription est directe comme dans le cas de la règle « les attributs d'une classe ne doivent pas être publiques ». Pour d'autres, un effort est nécessaire pour préciser certains aspects. Dans le cas de la règle « les méthodes ne doivent pas être grandes » par exemple, la notion de taille et le seuil de la taille doivent être précisés. Finalement pour une troisième catégorie, un effort de modélisation encore plus grand s'impose. C'est notamment le cas de la règle « une classe doit implémenter une seul abstraction » où la notion d'abstraction est relié au domaine de l'application et doit être approximée dans le code source.

Dans la suite de cette section, nous allons commencer par présenter un inventaire des règles de qualité définies dans la littérature. Nous donnons ensuite l'approche de modélisation. Cette approche s'appuie sur la logique floue pour représenter les seuils tels que nous le montrons dans la dernière sous-section.

5.1. Inventaire des règles de qualité

Les nombreuses années de pratique dans le développement logiciel ont permis l'émergence de nombreuses règles de qualité. Chaque règle modélise une situation qui peut avoir un impact sur une ou plusieurs caractéristiques de qualité telles que la fiabilité, la performance, la réutilisation ou la maintenabilité. Cependant, il n'existe pas dans la littérature une catégorisation permettant d'associer chaque règle à une caractéristique de qualité en particulier. En général, quand elles sont proposées, les auteurs de ces règles les associent à la qualité globale d'un programme.

Une autre particularité de ces règles est qu'elles sont définies à différents niveaux : architecture, structuration et détails d'implémentation. Elles sont de natures différentes et touchent à de degrés d'abstraction différents. Même le style des règles varie considérablement. Dans certains cas, les situations décrites sont considérées comme souhaitables, d'autres fois comme impératives et souvent comme des interdictions. Pour ajouter un niveau de complication supplémentaire, la terminologie utilisée est elle-même très variable.

Globalement, nous trouvons trois types de règles : les heuristiques de conception (Marinescu, 2002), les anti-patrons (Brown et al., 1998)(Riel, 1996) et les conventions de codage (des exemples peuvent être trouvées dans JCC[1]). Les heuristiques de conception sont des règles à suivre pour s'assurer de la qualité de la conception et éviter les défauts dans l'architecture. Nous donnons dans ce qui suit quelques exemples d'heuristiques :

- Toutes les données doivent être cachées à l'intérieur de leurs classes.
- Les classes utilisatrices d'une classe doivent dépendre de son interface publique, mais une classe ne doit pas dépendre de ses utilisatrices.
- La majorité des méthodes d'une classe doivent utiliser la majorité des attributs de cette classe la plupart du temps.
- Il ne faut pas créer des classes non nécessaires pour modéliser des rôles.

Les anti-patrons sont définis comme des solutions non souhaitables à des problèmes récurrents (Brown et al., 1998). Ces anti-patrons sont répertoriés dans des catalogues. Parmi eux on trouve :

- **Méthode jalouse :** Une méthode jalouse est une méthode qui semble plus intéressée par les attributs des autres classes que ceux définis dans sa classe.
- **BLOB ou God Class :** Une classe BLOB est une classe qui a tendance à centraliser le comportement d'une partie ou de la totalité du système.

[1] Java Code Conventions. Java.sun.com/docs/codeconv/

- **Refus d'héritage :** Quand une hiérarchie d'héritage est définie, on s'attend à ce que les classes utilisent les attributs et les méthodes héritées. Autrement, il est possible qu'il y ait un problème dans la conception de la hiérarchie des classes.
- **God Package :** Dans un package, on regroupe des classes qui ont des « affinités » l'une vers l'autre, autrement dit ces classes implémentent des fonctionnalités de la même catégorie. Si cette structuration n'est pas respectée, la cohésion du package est faible.

Finalement, les conventions de codage sont des règles relatives à l'écriture du code. Elles définissent des règles pour l'écriture des différentes structures de contrôles, les déclarations des variables, le choix des identificateurs, etc.

Comme nous pouvons le constater dans les exemples donnés plus haut, les règles peuvent être classifiées également en trois catégories selon le type d'information mis en jeu. Ces trois catégories sont :

1. Règles basées sur l'information structurelle. C'est-à-dire, les constructions syntaxiques du programme telles que les déclarations des classes, des méthodes et attributs, les relations d'association et d'héritage. C'est le cas par exemple de l'heuristique sur l'encapsulation des données.
2. Règles basées sur des métriques comme le montre par exemple l'anti-patron « méthode jalouse ».
3. Règles exprimant une notion abstraite. Elles nécessitent un effort de modélisation en termes de métriques et/ou d'information structurelle. Un bon exemple est l'heuristique sur la cohésion des classes.

5.2. Modélisation des règles de qualité

Comme nous l'avons déjà mentionné, de nombreuses règles sont exprimées en termes de métriques et d'information structurelle. En outre, l'information structurelle elle-même peut être représentée par des métriques. Si une règle, par exemple, requière l'existence d'une relation particulière entre deux éléments d'un programme, cette existence peut être modélisée par le fait qu'une métrique qui mesure la fréquence de la relation entre les deux éléments doit être non nulle. Par conséquent, une règle est représentée uniquement par un ensemble de conditions sur des métriques.

Cependant, lorsqu'une règle met en jeu une notion abstraite, il faut approximer cette notion par un ensemble de métriques. Ainsi, par exemple, pour la règle « une classe doit représenter une abstraction unique », les mesures de taille et de cohésion d'une classe peuvent être considérées comme des indicateurs du fait que la classe représente une abstraction unique ou non. Habituellement, si une classe est grande et peu cohésive (i.e. les méthodes utilisent peu ou pas les mêmes attributs), il y a de très fortes chances pour que plusieurs concepts ont été modélisés par une seule classe (Briand et al., 1998).

Pour permettre le choix ou la définition systématique des métriques pour chaque règle, nous utilisons l'approche GQM (*Goal-Question-Metric).* Cette approche consiste à définir des métriques à partir des buts et des questions qui leurs sont reliées (Basili et al., 1988). En effet, l'idée de cette approche est de d'abord identifier le but et éventuellement les sous buts de l'évaluation sous-entendue par la règle. Par la suite, les questions, dont les réponses permettent de savoir si les buts sont atteints, sont dérivées. Finalement les métriques qui permettent de répondre à ces questions sont déterminées.

Pour illustrer cette approche, considérons l'exemple de la règle correspondant à l'anti-patron BLOB (*GOD Class*). Comme nous l'avons défini plus haut, c'est une classe qui monopolise le comportement d'une partie ou de la totalité d'un système. Les caractéristiques de cette classe sont une grande taille, un degré de hiérarchisation faible (peu d'ancêtres et peu de descendants), les classes auxquelles elle est associée sont de petites tailles et de hiérarchisation faible également. Pour cette règle, le but (en termes de GQM) est de savoir si une classe est un BLOB ou pas. Deux questions permettront d'atteindre ce but :

1. Est-ce que la structure de la classe est une caractéristique du BLOB ?
2. Est-ce que la structure des classes associées est une caractéristique du BLOB ?

Pour répondre à la question 1, nous devons trouver/définir des métriques pour la taille de la classe et son degré de hiérarchisation. Ces mêmes métriques, quand appliquées aux classes associées, peuvent permettre de répondre à la question 2. Ainsi, nous pouvons schématiser la structure de la règle BLOB comme le montre la figure 3.

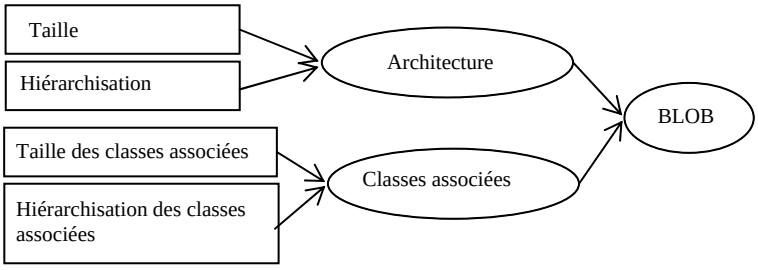

Figure 3. *Structuration de la règle BLOB*

La règle peut être formulée de manière informelle comme suit :

- Si la taille de la classe est grande et sa hiérarchisation est faible alors son architecture est caractéristique du BLOB.

- Si la taille des classes associées est petite et le degré de hiérarchisation de ces classes est faible alors ces classes peuvent être potentiellement des classes « structure de données ».

- Si l'architecture d'une classe est caractéristique du BLOB et que les classes associées sont potentiellement des classes « structure de données » alors cette classe est un BLOB.

Dans ces règles, la taille peut être mesurée par NOM et par le nombre de membres (attributs et méthodes). Le degré de hiérarchisation peut lui être mesuré par DIT, la profondeur dans l'arbre d'héritage.

Comme nous pouvons le constater dans cet exemple, ainsi que dans de nombreuses règles de qualité, l'évaluation des conditions à travers les métriques pose le problème de la définition des valeurs seuils. À partir de quelle valeur de la taille, nous pouvons considérer qu'une classe est grande ? La difficulté est d'autant plus grande que cette valeur a un impact sur la détection des anomalies. Afin de parer à cette difficulté, nous proposons l'utilisation de seuils flous. Dans la section suivante, nous allons présenter comment la logique floue nous permet de définir des règles à seuils flous.

5.3. Définition des seuils

Considérons la condition « la taille de la classe est grande ». On considère qu'une classe est grande si le nombre de membres est au moins égal à 20, une classe ayant 19 membres ne sera pas considérée comme grande alors qu'une avec 20 le sera. Cette coupure fixe ne permet raisonnablement pas d'envisager un bon taux de détection. Une approche plus réaliste consiste à définir des plages qui permettre d'introduire l'incertitude sur les seuils. Ainsi, par exemple, on pourra dire qu'une classe dont la taille est comprise entre 15 et 20 est possiblement grande alors qu'une autre avec une taille de 20 ou plus est grande. La logique floue permet d'exprimer cette notion en attribuant un degré de vérité compris entre 0 et 1 sur l'appartenance d'une valeur réelle à une étiquette floue comme le montre la figure 4.

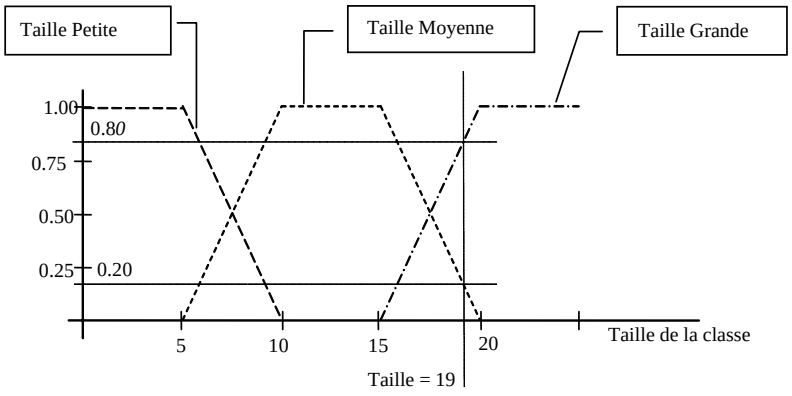

Figure 4. *Fonctions d'appartenances de la variable floue Taille.*

Si la taille est égale à 19, nous pouvons dire que la classe est grande avec un degré de vérité de 0.8 et moyenne avec un degré de vérité de 0.2.

En utilisant les étiquettes floues, une règle de qualité est modélisée par un système de règles floues. Dans ces règles les conditions s'expriment sous la forme de l'appartenance de valeurs de métriques à des étiquettes. La détection du BLOB peut-être exprimée par les règles suivantes :

***IF** NOM(c) Grande **&&** DIT(c) Faible **THEN** Architecturel_BLOB(c) Grand*

Pour chaque classe a de l'ensemble de classes associées A(c)

***IF** NOM(a) Petite **&&** DIT(a) Faible **THEN** Potentiel_classe_données(a) Grand*

***IF** Architecture_BLOB(c) Grand **&&** Potentiel_classe_données(A(c)) Grand **THEN** Classe BLOB*

En partant de là, l'évaluation des règles de qualité se fait en deux étapes :

– La flouefication (fuzzification) des valeurs des métriques en utilisant les fonctions d'appartenance.

– L'exécution du système de règle en utilisant un moteur d'inférence de règles floues. L'inférence permet de répercuter les incertitudes sur l'appartenance aux étiquettes floues à travers la chaine d'inférence.

Il y a plusieurs techniques pour définir les fonctions d'appartenance aux étiquettes floues. Dans notre projet, nous adoptons deux d'entre elles. La première est une technique automatique qui consiste en un algorithme de regroupement floue (*fuzzy clustering*) (Bezdek, 1981). La seconde est une technique semi-automatique qui consiste à dériver les fonctions d'appartenance à partir des distributions des valeurs des métriques considérées. Une description détaillée de cette technique incluant le choix des formes des fonctions est donnée dans l'article (Sahraoui et al, 2002).

La flouefication des valeurs seuils n'est pas toujours nécessaire. Dans certains cas, il y a un consensus sur ces valeurs. Par exemple, une règle populaire stipule que le nombre de paramètres d'une méthode ne doit pas dépasser 6. En considérant que *Par(m)* représente le nombre de paramètre de la méthode *m*, cette règle est transcrite dans notre langage de la manière suivante :

***IF** PAR(m) > 6 **THEN** Nombre Paramètre hors limite*

Par ailleurs, comme nous l'avons mentionné, certaines règles sont basées sur l'information structurelle qui peut être modélisée par des métriques. Ces métriques admettent deux catégories de valeurs possibles : l'absence (valeur numérique 0) et la présence (valeur numérique différente de 0). Par exemple, une règle connue stipule qu'il est fortement déconseillé d'avoir une relation de dépendance (invocation, association, agrégation, etc.) d'une classe vers une des classes de sa descendance. Cette règle peut être implémentée en utilisant une métrique, appelons la DEP, qui se base sur les primitives *dependance(c)* et *descendance(c)* représentant

respectivement l'ensemble des classes ayant une relation de dépendance avec la classe *c* et l'ensemble des sous-classes de *c*. DEP est égale au nombre de classes présentes dans les deux ensembles. Formellement :

$$DEP = \left| dependance(c) \cap descendance(c) \right|$$

La valeur de cette métrique est 0 s'il n'y a pas de relation de dépendance entre une classe et sa descendance. La règle peut donc être écrite comme suit :

IF *DEP(c) != 0* ***THEN*** *Dépendance inappropriée*

6. Travaux similaires

L'approche proposée dans cet article touche différents domaines et en particulier la définition de méta-modèles génériques pour la représentation du code, la collecte de métriques et la détection d'anomalies de conception.

Dans le contexte de l'évaluation de la qualité, de nombreux méta-modèles génériques ont été proposés afin de représenter le code source de différents langages (voir par exemple (Tichelaar et al., 1998) et (Harmer et al., 2002)). La principale spécificité de notre méta-modèle est la représentation explicite de la sémantique de certaines constructions permettant ainsi d'effectuer l'évaluation de la qualité sans une référence explicite aux langages.

Pour la collecte des métriques, certains travaux proposent des approches permettant la définition aisée de nouvelles métriques. En effet, dans (Mens, 2002), l'auteur définit un modèle de représentation de code source sous forme de graphe ainsi qu'un formalisme de définitions des métriques par la manipulation des nœuds et arcs du graphe. Les deux limitations de ce travail sont l'impossibilité d'exprimer certaines métriques usuelles et la difficulté d'utilisation du formalisme proposé.

La solution apportée par (Harmer et al., 2002), consiste à représenter le code source en utilisant une base de données relationnelle, et à utiliser des requêtes SQL intégrées dans un programme pour définir les métriques. Malgré, l'aspect déclaratif des requêtes, leur complexité est similaire à celle de l'utilisation d'un langage de programmation.

Dans (Baroni et al., 2003), les auteurs proposent l'utilisation d'OCL pour exprimer les métriques. OCL étant défini pour exprimer des contraintes sur des modèles UML, seul les métriques de conceptions sont implémentés. Les métriques qui mettent en jeu l'implémentation telles que celles mesurant certaines formes de couplage ne peuvent être directement exprimé avec OCL.

La détection d'anomalies par analyse de code source a été considérée dans plusieurs travaux. (Bär, 1998) propose d'exprimer les heuristiques sous la forme de requêtes en Prolog. Cette formalisation peut être appliquée principalement aux

heuristiques relatives aux propriétés structurelles d'un code source. Par contre l'information quantitative n'est pas prise en compte.

Dans (Marinescu, 2002), la détection de défauts dans la conception est basée sur la modélisation des heuristiques par un ensemble de conditions sur des métriques par rapport à des valeurs seuils. Une formulation basée sur la distribution de la métrique est utilisée pour déterminer ces valeurs seuils. Malgré le fait que cette approche permet de relativiser les seuils par rapport à la taille de l'application considérée, elle ne résout cependant pas complètement le problème des valeurs seuils dans le sens où les métriques sont quand même comparées à des valeurs fixes.

Un autre travail auquel nous nous comparons est celui de MeTHOOD (Grotehen et al., 1997). Il est constitué d'un catalogue d'heuristiques de conceptions, de règles de transformations et un mécanisme basé sur les métriques pour assister à trouver la meilleure solution à apporter à un problème détecté dans la conception. Les informations sur lesquelles cet environnement s'applique sont données par l'utilisateur à travers un éditeur dédié. Elles ne peuvent être extraites à partir du code source.

Par ailleurs, sur le plan des outils d'analyse des pratiques de codage, un exemple intéressant à citer est celui de PMD[2]. Ce dernier permet de détecter les violations d'un certain nombre de règle de codage. Il offre la possibilité à l'utilisateur de définir de nouvelles règles en écrivant du code Java selon la structure du pattern *visiteur*. Il permet également d'utiliser XPath[3] (*XML Path Language*) pour définir des requêtes ad hoc sur l'AST.

Dans toutes ces approches et outils, les règles de qualité sont implémentées directement dans les modules de détection à la différence de notre approche qui offre un langage de haut niveau à la fois pour définir les métriques et pour exprimer les règles. Par ailleurs, le fait que l'analyse est indépendante du langage de programmation permet de réutiliser les règles de qualité et de comparer la qualité de programmes écrits dans des langages différents.

7. Conclusion

Dans cet article, nous avons présenté une approche pour la détection de violations de règles de qualité utilisant l'analyse statique. Cette approche consiste en une solution intégrée et flexible. En effet, le langage de description permet d'ajouter facilement de nouvelles règles de qualité ou de modifier des règles existantes. L'utilisation de règles floues permet une démarche plus intuitive pour les utilisateurs lors de la modélisation des règles de qualité.

---

[2] PMD : pmd.sourceforge.net
[3] XPath : www.w3.org/TR/xpath

Le langage de description des métriques et des règles de qualité est composé d'un nombre restreint de construction syntaxique relativement simple afin qu'il puisse être utilisé par des non programmeurs. Cependant, la modélisation des règles de qualité (du moins les règles abstraites) exige une expertise dans le développement et la qualité des logiciels.

Dans sa version actuelle, notre environnement permet de représenter des programmes Java et C++. De nombreuses métriques ont été définies et un certain nombre de règles de qualité ont été modélisées parmi la cinquantaine de règles que nous avons recensées. Ces règles sont soit des heuristiques soit des anti-patrons. Actuellement, nous ne considérons pas les règles de bonnes pratiques de codage car elles sont difficilement modélisables en utilisant des métriques. Par exemple, si pour un langage, une règle préconise que les noms de méthodes ne doivent pas commencés par des majuscules, l'utilisation d'un analyseur syntaxique est plus appropriée. Cependant, comme future extension à notre approche, nous envisageons de définir des primitives spécifiques aux différentes constructions syntaxiques permettant de définir des règles de nature syntaxique.

Au stade de notre projet, nous n'avons pas encore conduit d'expériences contrôlées pour évaluer les performances de détection de notre approche. De telles expériences nécessitent une conception minutieuse car de nombreux facteurs externes peuvent influencer la qualité de la détection. Par exemple, si la détection utilisant une règle particulière est peu performante, ce résultat peut s'expliquer aussi bien par notre implémentation ou intrinsèquement par la validité de la règle elle-même. Ceci dit, le fait que nous avons réussi à implémenter un nombre de règles aisément combiné aux résultats très satisfaisants des quelques évaluations que nous avons conduites nous conforte dans les choix effectués.

8. Bibliographie

Baroni A., Brito e Abreu. F., « An OCL-Based Formalization of the MOOSE Metric Suite », *In Proc. of QUAOOSE'2003, at ECOOP'2003*, 2003.

Baroni A., Brito e Abreu F., « A Formal Library for Aiding Metrics Extraction », *International Workshop on Object-Oriented Re-Engineering at ECOOP'2003*, 2003.

Bär H., Ciupke O., « Exploiting Design Heuristics for Automatic Problem Detection », *In S. Ducasse and J. Weisbrod, editors, Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, number 6/7/98 in FZI Report, June, 1998.

Basili V.R., Rombach H.D., « The TAME project: Towards improvement-oriented software environments », *IEEE Transactions on Software Engineering*, 14(6), 1988.

Beugnard A., « Peut-on réaliser des composants avec un langage à objets? », *Langages et Modèles à Objets, LMO 2005*, Hermes, 2005.

Bezdek J. C., « *Pattern Recognition with Fuzzy Objective Function Algorithms* », Plenum Press, New York, 1981.

Briand L., Daly J., Wuest J.. « A Unified Framework for Cohesion Measurement in Object-Oriented Systems », *Empirical Software Engineering - An International Journal*, 1998.

Brown W.J., Malveau R.C., McCormick H.W., Mowbray T.J., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons. 1998.

Ciupke O., « Automatic Detection of Design Problems in Object-Oriented Reengineering », *In Technology of Object-Oriented Languages and Systems - TOOLS 30*, IEEE Computer Press, 1999.

Grotehen T., Dittrich K.R., « The MeTHOOD Approach: Measures, Transformation Rules, and Heuristics for Object-Oriented Design », Technical Report ifi-97.09, University of Zürich, Switzerland, 1997.

Harmer T.J., Wilkie F.G., « An Extensible Metrics Extraction Environment for Object-oriented Programming Languages », *Proceedings of 2nd IEEE Workshop on Source Code Analysis and Manipulation*, 2002.

Marinescu R., « Detecting Design Flaws via Metrics in Object-Oriented Systems », *In Proceedings of TOOLS*, IEEE Computer Society, 2001.

Marinescu R, Measurement and Quality in Object-Oriented Design, PhD Thesis. Politehnica University of Timisoara, 2002.

Mens T., « A Graph-Based Metamodel for Oriented-Oriented Software Metrics », *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

Riel A.J., *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

Sahraoui H., Boukadoum M., Chawiche H. M., Mai G., Serhani M. A, « A fuzzy logic framework to improve the performance and interpretation of rule-based quality prediction models for object-oriented software », In the proc. of the 26th Computer Software and Applications Conference (COMPSAC'02), 2002.

Tichelaar S., Demeyer S., « An Exchange Model for Reengineering Tools », *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, Serge Demeyer and Jan Bosch (Ed.), LNCS 1543, Springer-Verlag, July, 1998.