Université de Montréal

Un cadre d'application pour la visualisation des métriques orientées objet

par

Pascal Dufresne

Département d'Informatique et de Recherche Opérationnelle

Faculté des arts et des sciences

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

Mars 2003

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Un cadre d'application pour la visualisation des métriques orientées objets

présenté par :

Pascal Dufresne

a été évalué par un jury composé des personnes suivantes :

Président-rapporteur :	Petko Valtchev
Directeur de recherche:	Rudolf K. Keller
Membre du jury :	Jian-Yun Nie

Mémoire accepté le : 20 mars 2003

Sommaire

L'idée de mesurer les systèmes logiciels en est une vieille. Déjà, vers le milieu des années 60, des métriques furent proposées pour quantifier les caractéristiques des logiciels comme la complexité ou pour tenter de prédire les coûts de production.

Trente-cinq ans plus tard, bien qu'il soit maintenant établi que mesurer les logiciels est une bonne méthode pour en assurer la qualité, cette pratique a encore un taux d'acception très bas au niveau de l'industrie, et bien des gens concernés ne sont toujours pas convaincus de l'utilité des métriques. Les raisons qui expliquent ce piète état des choses sont multiples.

En premier lieu, bien que plusieurs chercheurs aient recommandé d'établir des fondements théoriques plus solides pour la discipline, à ce jour, la plupart des travaux publiés dans le domaine ont des fondements théoriques douteux. De plus, les métriques proposées ne sont souvent décrites qu'en langage naturel ce qui rend difficile leur validation empirique, la réutilisation des résultats par d'autres chercheurs ainsi que la communication entre ceux-ci. Finalement, bien que la visualisation soit maintenant une technique amplement utilisée en génie logiciel et ce, à toutes les étapes du cycle de développement, peu d'outils supportent la visualisation des résultats des métriques et quand la visualisation est supportée, seul un ensemble fixe de métriques est proposé ce qui laisse peu de place aux activités de recherche.

Cette recherche suggère des solutions aux problèmes énumérés ci-haut en proposant un cadre d'application pour le calcul et la visualisation des métriques orientées objets. Notre approche permet entre autres de mettre fin aux ambiguïtés relatives à la définition des métriques et, par conséquent, rend possible la comparaison et le partage des résultats des métriques. Aussi, en s'appuyant sur la théorie de la mesure et sur différents concepts du domaine de la psychophysique, notre cadre d'application définit des règles pour l'intégration de techniques de visualisation avec les métriques ainsi que pour la combinaison de plusieurs de ces techniques, le but étant de former

des visualisations efficaces des résultats des métriques. Finalement, un outil faisant usage de ce cadre d'application est présenté.

Mots-clés : métrique, visualisation, orienté objets, outil, cadre d'application

Table des matières

Liste des tableaux

Liste des figures

Liste des définitions

Liste des sigles et abréviations

API	Application Programming Interface
ASG	Abstract Semantic Graph
CASE	Computer Assisted Software Engineering
CIE	Commission International de l'éclairage
CSV	Comma Separated Variable
CVS	Concurrent Versioning System
FAMIX	FAMOOS Information Exchange Model.
FAMOOS	Framework-based Approach for Mastering Object-Oriented Software Evolution
HLS	Hue Lightness Saturation
HTML	Hyper Text Markup Language
ICSE	International Conference on Software Engineering
MDP	Metrics-Diagram-Perspective
MOOD	Metrics for Object Oriented Design
MOF	Meta Object Facility
MVC	Model-View-Control
OCL	Object Constraint Language
OMG	Object Management Group
RCR	Rétroconception Compréhension Réingenierie
RGB	Red Green Blue
SGBDOO	Système de gestion de base de donnée orienté objet
SPOOL	Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems
SQL	Structured Query Language
TCL	Tool Command Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Remerciements

Le dépôt de ce mémoire marque le fin d'une longue et enrichissante aventure universitaire. Au moment où je tourne la page sur cette importante période mon existence, je me dois de remercier une poignée d'individus qui ont joué, chacun a leur façon, un rôle déterminant dans cet épisode de ma vie. Certains m'ont conseillé, certains m'ont guidé, d'autres m'ont inspiré ou m'ont simplement donné leur amitié. Mais tous, sans exception, ont cru en moi. Certaines de ces personnes sont :

Mon directeur de recherche, monsieur Rudolf K. Keller.

Mes collègues du laboratoire GÉLO, de chez Bell Canada et du DIRO : Sébastien, Guy, Reinhard, Rui, Sarita, Greg, Hind, Jean-François, Bruno, Charles, François, Morad et plusieurs autres.

Mes parents, Denise et Lucien.

Et des amis qui m'ont particulièrement encouragé durant mon parcours: Anthony, Daniele, Sébastien, Binette.

Chapitre 1 : Introduction

1.1 Motivation

Lors de la conférence ICSE 2001 à Toronto, Mary Shaw présenta un modèle de maturation qui décrit de quelle façon les technologies en génie logiciel se développent et atteignent la maturité [73]. Selon ce modèle, une nouvelle technologie passe par six phases du moment de sa création au moment de son acceptation et de son utilisation générale. Ce processus dure typiquement de 15 à 20 ans [68]. L'idée de développer des mesures pour déterminer les caractéristiques des logiciels étant vielle de plus de 30 ans, on aurait pu croire que la discipline serait depuis longtemps arrivée à maturité et adoptée par tous. Or, à ce jour, l'utilisation des métriques demeure une pratique plutôt marginale dans l'industrie. Et bien que, ces dix dernières années, une hausse appréciable de l'utilisation des métriques ait été observée, dans beaucoup de cas c'est simplement parce que l'utilisation des métriques est une condition pour l'obtention de certaines certifications [33].

Pourtant, depuis trente ans, des milliers de métriques ont été proposées [92], et énormément de recherche a été fait dans le domaine. Alors comment peut-on expliquer cette stagnation ?

Simplement, presque tous les concepts attachés aux métriques sont soit ambigus ou, manquent de base théorique solide. De plus, lorsque des chercheurs tentent d'améliorer la situation, leurs travaux sont souvent ignorés. Par exemple, Fenton exprime son désarroi face au manque de rigueur qu'on retrouve dans plusieurs publications [32] :

> "It is over eleven years since DeMillo and Lipton outlined the relevance of measurement theory to software metrics. However, despite the important message in this work and related material, it has been largely ignored by both practitioners and researchers. The result is that much published work in software metrics is theoretically flawed.".

D'autres chercheurs [9] déplorent la confusion générale qui règne au niveau de la signification précise des attributs que l'on tente de mesurer :

> "Several different concepts are used in software measurement, e.g., size, complexity, cohesion, coupling. However, people used them in a very unrestricted way, so they are used inconsistently throughout the literature."

Churcher et Sheppard [19] émettent un avis similaire dans un article à propos des métriques de Chidamber et Kemerer [17]:

> "... we argue that it is premature to begin applying such metrics while there remains uncertainty about the precise definitions of many of the quantities to be observed and their impact upon subsequent indirect metrics."

Pire encore, le manque de standardisation rend extrêmement difficile la reproduction et la comparaison des résultats produits par différentes équipes de recherche. Souvent les publications ne contiennent même pas l'information nécessaire à la reproduction des résultats présentés comme en témoignent Churcher et Sheppard [20] :

> "This has led to attempts to assess and predict properties of software products and processes through the use of software metrics. Success has been limited by factors such as the lack of sound models, the difficulty of conducting controlled, repeatable experiments ... and the difficulty of comparing data obtained by different researchers or from different systems."

Plus loin, les auteurs posent un diagnostic plus général du problème:

> "..., the development and acceptance of software metrics as a discipline has been hampered by the indifferent quality of some experimental work and a lack of standardization of techniques and terminology"

Toutes ces ambiguïtés rendent difficile voir impossible la conception d'outils qui assisteraient réellement à la recherche dans le domaine des métriques. C'est pourquoi, typiquement, les outils qui supportent le calcul de métriques ne proposent qu'un ensemble pré-déterminé de métriques et ont donc un intérêt limité aux niveaux de la recherche. Pour des raisons similaires, le domaine des métriques a très peu profité de l'apparition récente d'une multitude d'outils CASE utilisant de façon intensive des techniques de visualisation avancées, ce qui aurait grandement aidé à l'acceptation et à la popularisation de la discipline.

L'objectif de ce mémoire est de présenter des solutions à plusieurs des problèmes chroniques énumérés ci-haut en développant un cadre d'application logiciel pour le calcul et la visualisation des métriques qui s'appuie sur des bases théoriques solides. Ce cadre d'application fournit une procédure standard pour définir, calculer et visualiser les métriques. Il incorpore également certains concepts de la théorie de la mesure ainsi que plusieurs résultats de la psychophysique expérimentale dans le but d'assister l'utilisateur dans la création de visualisations efficaces.

1.2 Contexte de ce travail

Ce travail a été réalisé dans le cadre du projet SPOOL [76]. Le projet SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) est une collaboration de l'équipe d'évaluation de qualité logicielle de Bell Canada et du groupe de génie logiciel GÉLO à l'université de Montréal. Ce projet s'inscrit au sein des activités initiées par le Consortium pour la recherche en génie logiciel (CRGL), un regroupement d'intervenants industriels, académiques et gouvernementaux qui vise l'élargissement du savoir en matière de génie logiciel au Canada.

L'objectif principal du projet SPOOL est d'identifier et de promouvoir les meilleures pratiques de conception et de développement qui contribuent à la qualité des systèmes logiciels orientés objet. La recherche s'articule autour de quatre axes complémentaires: les métriques de conception orientée objet, l'analyse de l'impact des changements, la traçabilité des transformations et le génie logiciel basé sur les patrons de conception. Les besoins de recherche ont mené au développement de l'atelier SPOOL pour faciliter les activités de rétroconception, de compréhension et de réingénierie.

L'architecture générale de l'atelier est conçue de manière à faciliter la création et l'intégration d'applications adaptées à des tâches spécifiques comme la rétroconception, la navigation, les analyses, le calcul de métriques et la visualisation. L'intégration est assurée à l'aide d'un dépôt central contenant les modèles des systèmes logiciels étudiés ainsi que les résultats des analyses. La métamorphose adoptée pour la modélisation des systèmes est dérivée du méta-modèle UML 1.1 et adaptée aux besoins concrets du projet SPOOL. Une description détaillée de l'environnement réalisé est fournie dans un livre publié récemment [72].

1.3 Structure de ce mémoire

Le Chapitre 2 présente différents aspects théoriques provenant de plusieurs domaines de recherche: les métriques, la théorie de la mesure, la psychophysique de

la visualisation, la visualisation d'informations, les cadres d'application et la méta-modélisation.

Le Chapitre 3 présente une évaluation de quelques outils commerciaux et académiques qui possèdent des fonctionnalité des visualisation et/ou de calcul de métriques

Le Chapitre 4 articule les problèmes auxquels nous nous attaquons dans ce mémoire.

Le Chapitre 5 décrit l'environnement SPOOL dont font usage les solutions logicielles créées dans la cadre de notre recherche.

MDP présente plusieurs solutions novatrices qui nécessitent la définition de plusieurs nouveaux concepts. Le Chapitre 6 s'affaire à décrire l'ensemble de ces nouveaux concepts.

Le Chapitre 7 est une documentation du cadre d'application MDP en tant que composante logicielle. Des diagrammes de classes sont présentés et les rôles des méthodes et des attributs important sont expliqués.

Le Chapitre 8 décrit l'outil Thycho-Metrics qui est basé sur le cadre d'application MDP. Cette description comporte plusieurs captures d'écrans.

Le Chapitre 9 résume brièvement ce qui a été accompli au cours de nos travaux et discute plus en profondeur de plusieurs aspects de ceux-ci.

Le Chapitre 10 est une courte conclusion qui examine les possibilités poursuite de notre recherche.

1.4 Contributions principales

La contribution majeure de ce travail est la définition d'un cadre d'application logiciel qui fournit la logique nécessaire à la création de visualisations efficaces pour les résultats des métriques orientées objet. À notre connaissance, c'est la première fois qu'un travail semblable est réalisé.

Les notions théoriques utilisées pour réaliser ce travail ont des sources diverses. Entre autres, des notions théoriques venant de la psychophysique de la visualisation, de la visualisation d'information et de la théorie de la mesure ont été utilisés dans ce mémoire. Une autre contribution de ce travail consiste à avoir trouvé de nouvelles applications de ces champs théoriques en génie logiciel.

La réalisation de ce travail a eu comme effet secondaire de nous forcer à trouver des solutions à plusieurs problèmes connexes à la visualisation des métriques. Plus particulièrement, il nous a fallu proposer une définition formelle et précise du concept de métrique dans le cadre de son environnement technologique. Nous croyons que c'est l'absence d'une telle définition qui explique que même après plusieurs décennies de recherche, les métriques ne sont toujours pas utilisées intensivement dans l'industrie. Un outil basé sur ce cadre d'application a également été conçu comme preuve de concept.

Chapitre 2 : Fondements théoriques

Ce chapitre contient un survol de certaines notions servant de base aux travaux présentés dans ce mémoire. Tout d'abord, les bases essentielles de la théorie de la mesure sont présentées ainsi que ses applications dans le contexte du génie logiciel. Ensuite, divers aspects des métriques seront brièvement présentés : les différents types de métriques, les métriques et l'orientation objet, les utilisations diverses des métriques et les modèles de qualité. Nous poursuivons avec un exposé sur la visualisation d'information et les considérations psychophysiques qui s'y rattachent. Ensuite, un survol rapide des caractéristiques des cadres d'applications orientés objet sera fait. Finalement, les notions de modèle et de méta-modèle seront discutés.

2.1 Métriques

2.1.1 Théorie de la mesure

Les bases

Il est unanime que la théorie de la mesure est un fondement théorique nécessaire à la discipline qui consiste à mesurer les logiciels [32]. Nous exposerons donc l'essentiel de celle-ci. Un ouvrage de Roberts [69] ainsi qu'un rapport technique de Briand [9] ont été utilisés pour rédiger notre aperçu.

Définition 1 : Système relationnel

Soit $R_1, R_2, \ldots, R_p$ des relations (pas nécessairement binaires) sur le même ensemble A, et $\mathbf{o_1}, \mathbf{o_2}, \ldots, \mathbf{o_q}$ des opérations binaires sur A. Le (p+q+1)-tuple

$$\Re = (A, R_1, R_2, \ldots, R_p, \mathbf{o_1}, \mathbf{o_2}, \ldots, \mathbf{o_q})$$

est appelé un *système relationnel*. Ici, les relations $R_1, R_2, \ldots, R_p$ ont le sens mathématique usuel soit, un sous-ensemble d'un produit cartésien multiple. Par

exemple si la relation R est de dimension n, elle est en fait un sous-ensemble de A^n. Les opérations binaires o_n doivent être vues comme étant des fonctions f_n définies comme suit :

$$f_n : A \times A \rightarrow A$$

(Pour plus de détail, voir [69] section 1.8). En ce qui a trait à la théorie de la mesure, deux types de système relationnel sont d'intérêt : les *systèmes relationnels empiriques* qui décrivent les objets que l'on désire mesurer et les *systèmes relationnels formels* qui décrivent les objets mathématiques et formels que l'on désire utiliser pour représenter les caractéristiques de nos objets.

Définition 2 : Système relationnel empirique

Un *système relationnel empirique* $\Im$ est un système relationnel tel que

$$\Im = (E, R_1, R_2, \ldots , R_p, o_1, o_2, \ldots , o_q)$$

et

E est un ensemble d'objets *empiriques*,

R_n sont des relations *empiriques* sur E et

o_n sont des opérations binaires sur l'ensemble E.

Définition 3 : Système relationnel formel

Un système relationnel formel $\wp$ est un système relationnel tel que

$$\wp = (F, S_1, S_2, \ldots , S_p, \bullet_1, \bullet_2, \ldots , \bullet_q)$$

et

F est un ensemble d'objets *formels*,

S_n sont des relations sur F

$\bullet_n$ sont des opérations binaires sur l'ensemble F

Le but d'un système relationnel empirique est de décrire l'ensemble d'objets E dans le contexte d'un attribut que nous tentons de mesurer. Ces objets sont dits *empiriques* car certaines de leurs caractéristiques ne peuvent pas être décrites de façon formelle et

précise. En physique, ces objets peuvent aussi bien être des particules élémentaires que des étoiles ou des champs magnétiques. En sciences sociales, ces objets seront souvent des humains. En génie logiciel, ces objets peuvent être des éléments du code source (classes, méthodes, expressions), les processus utilisés pour développer le logiciel ou quoi que ce soit ayant un rapport quelconque avec un logiciel.

Les connaissances empiriques que nous avons des attributs des objets de l'ensemble E sont décrites de façon informelle dans les relations R_i et dans les opérations binaires $\mathbf{o_i}$. Ces connaissances ne sont pas le résultat de mesure (on assume qu'il n'y a pas encore de mesure pour ces attributs), mais bien le résultat d'observations empiriques.

Le système relationnel formel sert à représenter un attribut spécifique des objets de l'ensemble E mais, dans un monde mathématique formel. Plusieurs considérations entrent en ligne de compte dans le choix d'un système relationnel formel. Premièrement, les caractéristiques de l'ensemble d'objets empiriques E doivent être prises en considération. Par exemple, l'échelle des nombres naturels ne peut être utilisée pour mesurer la taille puisque la taille est une quantité continue. Il sera aussi nécessaire de trouver une relation formelle (<,>,=) pour exprimer chaque relation du système empirique ainsi qu'une opération binaire mathématique formelle (+,-) pour exprimer chacune des opérations binaires du système empirique. En général, les systèmes relationnels formels qui nous concernent sont des construits mathématiques bien connus comme l'ensemble des nombres réels, l'ensemble des nombres naturels ou l'ensemble des nombres entiers. Les relations seront aussi les plus communes (>,<,=) et de même pour les opérations binaires (addition, soustraction, multiplication, division).

Nous donnons maintenant un exemple de mesure en génie logiciel. Dans le domaine du génie logiciel orienté objet, une des mesures les plus communes est la mesure de la complexité. Plus précisément, supposons que l'on s'intéresse à la complexité d'un module. Dans ce cas, l'une des relations *C* du système empirique formel pourrait être définie de la façon suivante:

$$(a, b) \in C \Leftrightarrow \text{a est « plus complexe » que b}$$

Bien qu'il soit très clair que certains modules sont « plus complexes » que d'autres, il n'existe aucune définition formelle de ce concept et au niveau du code source lui-même, les causes exactes de la complexité d'un logiciel sont encore mal comprises. Dans ce cas-ci, un expert des logiciels peut être utilisé pour établir la relation empirique. En regardant le code source de deux modules, il peut déterminer lequel est plus complexe *selon lui*.

Le concept de complexité en génie logiciel est un attribut très abstrait. C'est pourquoi, on tente toujours de définir un système relationnel empirique qui fasse l'unanimité. Par exemple, [92] en s'appuyant sur des concepts avancés de la théorie de la mesure, suggère que toute mesure de la complexité doit être additive. Cela signifie que la complexité d'un morceau de code P composé de deux bloques distincts B_1 et B_2 doit être égale à la somme des complexités de B_1 et B_2. Par conséquent, le système relationnel empirique doit inclure une opération de *concaténation*, c'est-à-dire il doit définir comment mettre deux morceaux de code source, des modules par exemple, ensemble. Pour sa part, le système relationnel formel inclura l'opérateur d'addition. De son coté Briand, maintient que cet axiome pose des conditions trop strictes pour la définition des mesures de complexité [9].

En se rangeant du coté de Zuse, pour les systèmes relationnels empiriques et formels de la mesure de la complexité des modules, on obtient respectivement :

$$\Im = (M, C, o) \qquad \text{et} \qquad \wp = (Re^{\geq 0}, >, +) ,$$

où M est l'ensemble de tous les modules pouvant exister, C une relation empirique imposant un ordre total strict basé sur la complexité sur l'ensemble M, et o un opérateur de concaténation qui définit comment joindre deux modules pour n'en faire qu'un seul.

Une fois les deux systèmes relationnels définis, il faut trouver une façon de projeter le système relationnel empirique $\Im$ dans le système relationnel formel $\wp$. C'est ce qu'on appelle trouver une *mesure*. Il est maintenant temps de définir formellement le concept de mesure.

Définition 4 : Mesure

Soit

$$\Im = (E, R_1, R_2, \ldots , R_p, \mathbf{o}_1, \mathbf{o}_2, \ldots , \mathbf{o}_q)$$

un système relationnel empirique et

$$\wp = (F, S_1, S_2, \ldots , S_p, \bullet_1, \bullet_2, \ldots , \bullet_q)$$

un système relationnel formel où r_i, la dimension de R_i, est égale à la dimension de S_i $\forall i$. Une mesure μ, est un homomorphisme entre $\Im$ et $\wp$. C'est-à-dire que μ est une fonction μ: $E \rightarrow F$ tel que $\forall$ i et $\forall$ $e_1, e_2, \ldots , e_{ri} \in E$,

$$R_i (e_1, e_2, \ldots , e_{ri}) \Leftrightarrow S_i (\mu(e_1), \mu(e_2), \ldots , \mu(e_{ri}))$$

et $\forall$ j, $\forall$ b,c $\in$ E,

$$\mu(b \; \mathbf{o}_j \; c) = \mu(b) \bullet_j \mu(c)$$

Donc, on s'attend d'une mesure qu'elle préserve les relations empiriques. Dans le cas de la mesure de la complexité décrite ci-haut, on voudra que notre mesure μ soit telle que

$$a \; C \; b \Leftrightarrow \mu(a) > \mu(b).$$

On voudra également que les opérations binaires soient préservées. Dans le cas de la mesure de la complexité, l'opérateur **o** devra être préservé:

$$\mu (a \; o \; b) = \mu(a) + \mu(b)$$

Si les relations et les opérations binaires sont satisfaites, on dit que la mesure satisfait la *condition de représentation* (voir [69] chapitre 2.4). Donc, trouver une mesure pour un attribut quelconque d'une classe d'objets empiriques c'est trouver un homomorphisme depuis le système relationnel empirique qui décrit la classe d'objets vers un système relationnel formel approprié. Concrètement, il s'agira de trouver une façon d'observer l'attribut que l'on désire mesurer d'une façon qui rende possible la définition d'une fonction vers l'ensemble d'objets mathématiques choisi et ce, de

façon à ce que toutes les relations empiriques ainsi que toutes les opérations binaires soient respectées.

Trouver une mesure se fait généralement par un processus d'essai et d'erreur. On propose tout d'abord une mesure, ensuite on vérifie si les relations et les opérations binaires empiriques sont respectées. Si elle le sont, on peut dire que, théoriquement, on a découvert une mesure pour l'attribut en question. Cependant, puisque les relations et les opérations binaires empiriques sont souvent informelles, il n'y a pas de façon formelle de valider une mesure. Dans le cas de la mesure de la complexité par exemple, un premier spécialiste peut estimer qu'un module A est plus complexe qu'un module B et ce, en accord avec une certaine mesure proposée, alors qu'un autre spécialiste peut estimer que B est plus complexe que A. La validation d'une mesure est donc souvent une activité subjective.

Type d'échelles *(scale types)*

En général, il n'existe pas de mesure unique pour un couple de systèmes relationnels donné. Les mesures en degré Fahrenheit et Celsius sont des mesures distinctes qui satisfont toutes deux le problème de la mesure de la température. De même, pour la mesure du poids, les mesures en kilogrammes et en livres sont deux mesures acceptables. Nous devons maintenant introduire la notion de type d'échelles qui sera utile pour faire la visualisation des métriques. Les quatre définitions suivantes sont nécessaires pour définir la notion de type d'échelles.

Définition 5 : Échelle (Scale)

On appelle une *échelle* un 3-tuple ($\mathfrak{I}$, $\wp$, μ) où $\mathfrak{I}$ est un système relationnel empirique, $\wp$ un système relationnel formel et μ une mesure (homomorphisme) de $\mathfrak{I}$ vers $\wp$.

Définition 6 : Transformation admissible d'échelle

Soit ($\Im$, $\wp$, μ) une échelle et $\phi : \mu(\Im) \rightarrow \wp$ une fonction. Si (ϕ o μ) est une mesure (homomorphisme) de $\Im$ vers $\wp$, on dit que ϕ est une *transformation admissible* de l'échelle ($\Im$, $\wp$, μ).

Définition 7 : Échelle régulière

On dit d'une échelle ($\Im$, $\wp$, μ) qu'elle est *régulière* si et seulement si $\forall$ échelle ($\Im$, $\wp$, g), $\exists$ une transformation admissible $\phi : \mu (\Im) \rightarrow \wp$ tel que g = ϕ o μ .

Définition 8 : Représentation régulière

Soit $\Im$ un système relationnel empirique et $\wp$ un système relationnel formel. On dit que $\Im \rightarrow \wp$ est une *représentation régulière* si toutes les échelles ($\Im$, $\wp$, μ) possibles sont régulières.

Si une représentation $\Im \rightarrow \wp$ est régulière, la classe des transformations admissibles sur une échelle ($\Im$, $\wp$, μ) quelconque défini un *type d'échelle*. Le type d'une échelle nous permet donc de définir toutes les mesures équivalentes à une mesure μ. Le Tableau 1 présente les échelles les plus communes. L'idée de définir des types d'échelle d'après la classe des transformations admissibles est due à S.S. Stevens [78]. Bien que cette classification ait subi plusieurs critiques [84], elle nous sera suffisante.

Types d'échelles			
Échelle	**Transformations admissibles φ**	**Opérations de base possibles**	**Exemples**
Nominal	Transformation injective $\varphi(x) = \varphi(y) \Leftrightarrow x = y$	Vérification d'égalité	Numéro sur uniforme de joueurs
Ordinal	Strictement monotone $x \geq y \Leftrightarrow \varphi(x) \geq \varphi(y)$	Plus que, moins que	Préférence, danger de feux de forêt
Intervalle	Linéaire positive $\varphi(x) = \alpha x + \beta$ où $\alpha > 0$	Vérification de l'égalité des différences	Température (Fahrenheit, Celsius)
Ratio	Multiplication $\varphi(x) = \alpha x$ où $\alpha > 0$	Vérification de l'égalité des ratios	Température (Kelvin)
Absolue	Identité $\varphi(x) = x$	Les opérations possibles sur les nombres naturels	Compter le nombre d'objets

Tableau 1: Les types d'échelle

La troisième colonne du Tableau 1 est additive : les opérations possibles sur les échelles de type Nominal et Ordinal sont aussi possibles sur les échelles de type Intervalle et ainsi de suite.

Une échelle est dite *Nominale*, lorsque la mesure ne fait que distribuer les objets empiriques dans différentes catégories qui ne sont pas ordonnées. Dans ce cas la seule relation définie est la vérification d'égalité. On peut se servir des nombres naturels $\Im$ = (N, =) ou des nombres réels $\Im$ = (Re, =) pour une échelle nominale. La numérotation des joueurs d'une équipe sportive en est un exemple : les numéros ne sont réellement que des identificateurs pour distinguer les joueurs, leur ordre n'a aucune signification. Puisque les objets ne sont pas ordonnés, on peut aussi bien se servir d'un ensemble d'identificateurs prédéfinis. Par exemple, en génie logiciel orienté objet, une mesure de « L'origine d'une méthode » pourrait avoir un système relationnel formel ressemblant à celui-ci.

$$\Im = (\{\text{Redéfini, Hérité, Défini}\}, =)$$

Une échelle est dite *Ordinale*, lorsque la mesure distribue les objets empiriques dans des catégories distinctes et totalement ordonnées mais, sans que l'on puisse dire quoi que ce soit sur la distance entre ces catégories. Ce genre d'échelle est souvent utilisé dans les sondages d'opinion publique où des échelles du genre

$$\Im = (\{\text{Très insatisfait, insatisfait, satisfait, Très satisfait}\}, >)$$

appelées *échelles de Likert* sont communes. Ici il n'est pas nécessaire de mentionner la relation d'égalité puisqu'on peut la déduire depuis la relation plus-grand-que (>). On peut également utiliser les nombres réels ou les nombres naturels et profiter de leur ordre total. L'indice de danger de feux de forêt (de 1 à 5) est un exemple d'utilisation des nombres naturels pour une échelle ordinale à condition que l'on n'attribue pas de signification à la distance entre les niveaux.

Une échelle est dite *Intervalle*, lorsque la mesure distribue les objets empiriques dans des catégories dont les distances relatives sont connues. C'est à partir de ce *type d'échelles* qu'on peut parler de mesure quantitative au sens usuel du terme. Ici la distance entre les objets formels doit être connue ce qui rend nécessaire l'utilisation de l'ensemble des réels ou d'un sous-ensemble de celui-ci. Les échelles de degrés Celsius et Fahrenheit sont des exemples. Les intervalles entre les degrés correspondent à des augmentations égales de la hauteur d'une colonne de mercure. La position du zéro est arbitraire et est choisie de façon à être pratique (point de fusion et ébullition de l'eau). On peut passer d'une échelle à l'autre en appliquant la forme $\varphi(x) = \alpha x + \beta$ où $\alpha > 0$. Pour la conversion de l'échelle Celsius vers Fahrenheit $\alpha = 9/5$ et $\beta = 32$.

Une échelle est dite *ratio*, si la mesure assigne les objets empiriques à des objets formels dont les ratios peuvent être comparées. Puisque les ratios, doivent demeurer les mêmes d'une échelle à l'autre, il se doit d'y avoir un zéro commun à toutes les échelles de la représentation. Les seules transformations d'échelles permises sont donc $\varphi(x) = \alpha x$ où $\alpha > 0$. Les échelles de type ratio sont les plus courantes en physique. La force, la masse, la température (en degré Kelvin) et la charge électrique sont toutes représentées sur des échelles de type ratio. Plusieurs échelles existent pour chacun de ces attributs, et les valeurs peuvent toujours être converties d'une échelle à une autre par multiplication par un facteur de conversion (1kg = 2.2 lbs).

Et finalement, une échelle est dite *absolue* si la mesure est la seule appropriée. La seule transformation permise est donc l'identité. Toutes les mesures de type absolue sont en fait le comptage d'éléments d'un ensemble quelconque. Toutes les opérations

arithmétiques permises sur les nombres naturels sont aussi permises sur les échelles de type absolue.

2.1.2 Les métriques

Définition 9 : Métrique (en tant que mesure)

Plusieurs définitions du mot *métrique* ont été proposées (voir [92] p.15). En nous appuyant sur la théorie de la mesure, nous dirons simplement qu'*une métrique est une mesure pour un certain attribut ayant un rapport quelconque avec les logiciels*, le mot *mesure* ayant la signification de la Définition 4.

Classification

En génie logiciel, ce n'est pas seulement le code source qui peut être l'objet d'une mesure. Fenton identifie trois grandes catégories de métriques [34]. La classification est faite selon la nature de l'objet qui est mesuré. Elle est supportée par Zuse [92] et généralement acceptée.

Catégorie	Entité mesuré
Produit	spécification, design, code source, données des tests, documentation, ...
Processus	processus de spécification, de conception, de conception détaillée, des tests, ...
Ressource	personnel, équipe, logiciel tiers-partie utilisé, matériel utilisé, environnement de travail, ...

Tableau 2: Les catégories de métriques selon Fenton

Le Tableau 2 décrit comment les entités sont classées. Les métriques de *processus* sont les métriques qui mesurent les attributs d'une activité quelconque liée aux logiciels. Tout document, toutes entités produites durant ces activités seront mesurés avec des métriques de *produit* alors que les ressources utilisées durant ces activités seront mesurées à l'aide de métriques de *ressource*. Dans ce mémoire, nous nous intéressons à un sous-ensemble bien précis de métriques soit les métriques de design et de code source qui font partie de la catégorie *produit*.

Au niveau des attributs eux-mêmes, Fenton fait une distinction entre les attributs internes et les attributs externes. Les attributs internes sont définis comme étant des

attributs qui peuvent être mesurés en fonction de l'entité mesurée soit un processus, un produit ou une ressource uniquement, alors que les attributs externes sont ceux qui ne peuvent être mesurés qu'en rapport avec l'environnement dans lequel ils se trouvent. La taille d'un logiciel est un attribut interne car elle peut être mesurée directement en comptant le nombre de lignes de code par exemple. D'un autre coté, le temps de démarrage d'un logiciel doit être considéré comme un attribut externe car il ne dépend pas seulement du logiciel lui-même mais de plusieurs autres facteurs comme la vitesse du processeur et la mémoire vive de l'ordinateur utilisé. Le Tableau 3 présente quelques attributs internes et externes du design et du code source.

Entité	Attributs	
	Internes	Externes
Design	Taille, réutilisation, modularité, couplage, cohésion, fonctionnalité	Qualité, complexité, maintenabilité
Code Source	Taille, réutilisation, modularité, couplage, fonctionnalité, complexité algorithmique	Fiabilité, utilisabilité, maintenabilité

Tableau 3: Exemples d'attributs internes et externes pour le design et le code source

Finalement, au niveau des mesures, Fenton fait une distinction entre les mesures directes et indirectes. Une mesure directe d'un attribut est une mesure qui ne dépend d'aucun autre attribut. Par exemple, la longueur d'un objet concret peut être mesurée sans aucune référence à ses autres attributs. La mesure de la densité d'un objet, elle, est considérée comme une mesure indirecte car, en général, elle est le résultat du quotient entre une mesure de poids et une mesure de volume.

Objectifs des métriques

Il y a deux grands objectifs à l'utilisation des métriques, l'évaluation et la prédiction. Souvent, on s'intéresse aux attributs qui existent déjà. Cela est utile pour évaluer l'état actuel des choses. Cependant, dans plusieurs cas, on voudra prédire la valeur d'un attribut d'un objet qui n'existe pas encore. Pour cela il faudra mettre au point un *modèle de prédiction* qui explique comment la mesure d'attributs qui existent présentement peut être utilisée pour prédire la valeur d'un attribut d'un objet qui n'existe pas encore. Par exemple, le modèle COCOMO II de Barry Boehm permet d'estimer le coût de production d'un logiciel en fonction du type de logiciel et de sa

taille [13]. Mettre au point un tel modèle peut être une tâche laborieuse, mais c'est souvent la seule façon de recueillir des mesures vraiment utiles. Par exemple, après la complétion d'un projet il est très facile de mesurer son coût, il suffit d'additionner toutes les dépenses faites. Cependant, cette mesure n'est pas utile puisque si le coût s'avère trop élevé, il est trop tard pour changer quoi que ce soit. Construire un modèle de prédiction fiable est difficile mais cela permet d'identifier les problèmes pendant qu'on peut encore faire quelque chose. Puisque les mesures de prédiction utilisent les mesures d'autres attributs, elles sont nécessairement des mesures indirectes.

2.1.3 Les métriques orientées objet

Origine

L'apparition des langages orientés objet à causé un certain bouleversement dans la communauté des métriques. Il faut bien le dire, la programmation objet n'est pas simplement un ensemble de fonctionnalités ajoutées aux langages de programmation fonctionnels, c'est une approche complètement différente pour concevoir des logiciels [14]. On ne peut donc pas s'attendre à ce que les métriques conçues pour les langages fonctionnels traditionnels puissent être appliquées directement aux langages orientés objet. Cet avis est exprimé entre autres par Churcher et Sheppard [20], par Chidamber et Kemerer [17] et plus récemment par Bansiya et Davis [4] qui eux, considèrent que les concepts orientés objet comme l'abstraction et l'encapsulation doivent eux-mêmes être mesurés. En fait, le besoin pour des métriques *orientées objet* fut identifié dès 1991 [20].

Chidamber et Kemerer

La première tentative sérieuse de définir des métriques orientées objets ayant une certaine base théorique fut faite par Chidamber et Kemerer[17]. En s'appuyant solidement non seulement sur la théorie de la mesure, mais aussi sur certaines notions philosophiques et sur les propriétés de Weyuker [89], ils proposent six métriques fortement liées aux concepts orientés objet. Depuis la parution de cet article, la plupart des publications liées aux métriques orientées objet font mention de cet ouvrage quand elles ne sont pas directement basées dessus.

Conformément aux principes ontologiques utilisés, les métriques sont définies indépendamment des détails de l'implémentation, et aucune tentative n'est faite pour définir ces métriques en fonction d'un langage de programmation en particulier.

Bien que cette publication ait servi de base à presque tous les travaux ultérieurs portant sur les métriques orientées objet, elle a reçu un grand nombre de critiques. Premièrement, au niveau de l'utilisation de la théorie de la mesure, Hitz et Montazeni [41] considèrent que les systèmes relationnels empiriques proposés sont incomplets car ils n'incorporent pas toutes les idées intuitives qu'on se fait a propos des attributs mesurés. Ces auteurs démontrent également qu'au moins une des métriques, LCOM, ne préserve même pas les relations empiriques définies pour l'attribut par les auteurs de l'article. Finalement ils remarquent que leurs auteurs ont substitué la condition de représentation pour une validation basée sur la conformité avec les propriétés de Weyuker[89].

De leur coté, Churcher et Sheppard [19] croient qu'il serait futile de tenter de définir des métriques indirectes complexes tant qu'il persistera une certaine ambiguïté quant à la définition précise des mesures directes à effectuer. Ils font remarquer que, bien que Chidamber et Kemerer prétendent avoir développé un ensemble de métriques indépendantes du langage de programmation utilisé, l'application de ces métriques à un langage en particulier nécessite un nombre de décisions tout à fait subjectives ce qui rend incomparable les résultats obtenus par des personnes différentes.

Utilisations diverses des métriques orientés objets

En plus des deux objectifs généraux des métriques, la vérification et la prédiction, les métriques orientées objets ont été utilisées pour effectuer toutes sortes de tâches.

L'une des caractéristiques des systèmes légataires est qu'il n'existe pas ou peu de documentation sur la conception initiale du système. Il est donc difficile de maintenir ces systèmes. Antoniol [1] utilise les métriques pour identifier les composantes de conception dans les systèmes légataires.

Le *refactoring* [36] est une activité de maintenance qui vise à améliorer la structure interne d'un système sans changer ses fonctionnalités. Plusieurs outils visant à identifier les cibles de *refactoring* ont été conçus. La plupart d'entre eux utilisent des métriques pour identifier ces cibles [74][22][79].

Dans d'autres circonstances on voudra évaluer la *changeabilité* d'un système, c'est-à-dire l'aisance avec laquelle un système s'adapte à des changements dans le code source. Diverses métriques peuvent être utilisées pour faire cette évaluation [46].

2.1.4 Les modèles de qualité

Le génie logiciel s'intéresse à tous les aspects du logiciel : les ressources nécessaires, les processus utilisés pour la spécification, la conception, la maintenance et le support matériel et logiciel utilisé. Mais en ce qui concerne le produit lui-même, le but est qu'il soit de qualité, c'est-à-dire qu'il soit bien conçu. L'objectif ultime des métriques de produit est donc d'évaluer la qualité d'un logiciel ou de prédire quelle sera sa qualité. Ceci exige qu'on définisse la qualité de façon à ce qu'elle puisse être mesurée ce qui est très problématique, car on peut voir la qualité sous plusieurs perspectives. L'utilisateur du produit dira que le produit est de qualité s'il remplit ses besoins alors qu'un manufacturier dira que son produit est de qualité quand son processus de production est certifié. Une façon plus objective d'évaluer la qualité d'un produit consiste à évaluer ses caractéristique internes[49]. Il faut seulement s'assurer que la qualité interne du produit entraînera sa qualité externe.

Plus concrètement, dans le monde des logiciels, pour accomplir cela, il faudra développer un *modèle de qualité*. Un modèle de qualité typique utilise l'approche par décomposition, il comprend un ensemble de facteurs de qualité, c'est-à-dire les attributs externes du produit qui déterminent sa qualité. Pour chacun de ces attributs externes, un ensemble d'attributs internes est décrit, ainsi qu'un ensemble de métriques pour mesurer ces attributs internes. La Figure 1 illustre ce type de modèle de qualité. Bien sûr, le modèle doit expliquer de quelle façon chacun des facteurs entraîne la qualité. Il doit également présenter une logique dans le choix des attributs

internes et des métriques, une logique qui, préférablement, s'appuiera sur la théorie de la mesure.

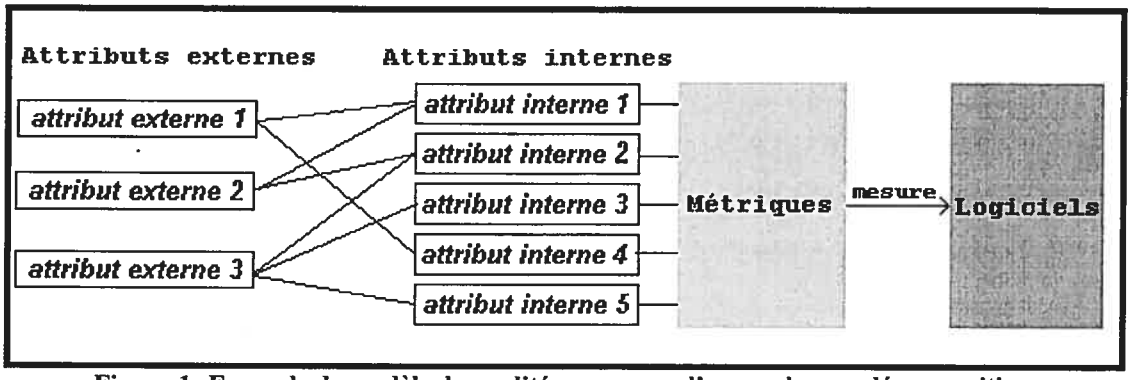


Figure 1: Exemple de modèle de qualité conçu avec l'approche par décomposition

Deux des premiers modèles de qualité créés utilisent l'approche par décomposition[59][12]. Plus récemment, l'ISO proposa une définition standard de la qualité pour en déduire la mesure [43]. Le modèle est composé de 6 facteurs de qualité standard qui sont définis de façon informelle. Pour chacun de ces facteurs (attributs externes) un ensemble de sous-facteurs (attributs internes) est proposé. Finalement, pour chaque sous-facteur un indicateur est fourni pour aider à la définition des métriques utilisées pour mesurer le sous-facteur.

Le point faible de tous les modèles proposés jusqu'à présent est le manque de logique derrière la plupart des choix effectués pour construire le modèle. Peu d'explication est donné quant aux choix des facteurs de qualité et de leurs sous-facteurs. De plus, la façon exacte par laquelle les métriques doivent être appliquées et combinées n'est, en général, pas décrite [49]. Bien que certains modèles alternatifs ont été proposés pour répondre à ces problèmes [26], ils n'ont été que très peu adoptés.

2.2 La visualisation

2.2.1 Aspects psychophysiques de la visualisation

La psychophysique est l'étude des rapports entre les stimulus physiques et les sensations qu'ils provoquent. Le système visuel de l'être humain a été le sujet de nombre d'études et plusieurs résultats expérimentaux sont pertinents au domaine de la visualisation en génie logiciel.

L'approche présentée dans cette recherche consiste à visualiser les résultats des métriques de différentes manières. L'une de ces manières consiste à utiliser des *glyphs*. Un glyph est un objet graphique quelconque dessiné de façon à pouvoir communiquer plusieurs données. Nous sommes donc intéressés à savoir quelles sont les différentes façons dont un glyph peut communiquer de l'information et, quelles sont les règles à appliquer pour qu'une visualisation utilisant des glyphs soit la plus efficace possible.

Les travaux de Anne Treisman [81][80][82] sont d'un intérêt particulier parce qu'ils couvrent ces deux aspects à la fois. Elle définit une *dimension* comme étant un ensemble de valeurs provenant d'un même stimulus et qui s'excluent mutuellement. La couleur à elle seule peut exprimer jusqu'à trois dimensions. Une valeur à l'intérieur d'une dimension est appelée *propriété*. Les expériences conduites par Treisman consistent à présenter une série d'images à un groupe de sujets. En général, les images contiennent un ensemble de glyphs qui peuvent être différents sur une ou plusieurs de leurs dimensions. On demande aux sujets de faire la recherche d'une figure en particulier, appelée *cible*, selon les spécifications de certaines de ses dimensions. Les autres figures de l'image sont appelées les *distractions*.

Processus attentifs et pré-attentifs

Comme plusieurs autres chercheurs, Treisman fait la différence entre les processus pré-attentifs (*pre-attentive processes*) et les processus attentifs (*attentive processes*) pour caractériser le mécanisme par lequel les sujets identifient la cible. Les processus pré-attentifs sont en général très rapides, ils nécessitent environ 100ms ou moins. Ceci suggère que le traitement de toute l'image se fait d'un seul coup par le cerveau, tous les glyphs de l'image étant traités en parallèle. Les processus attentifs, eux, nécessitent que l'attention du sujet soit portée séquentiellement sur chaque figure. Le temps utilisé par le sujet pour trouver la cible augmentera donc avec le nombre de distractions dans l'image alors que dans le cas des processus pré-attentifs l'identification de la cible se fera en un temps qui dépend peu du nombre de distractions. Bien qu'initialement une telle dichotomie fut largement utilisée, des

expériences plus récentes ont démontré qu'il existe plutôt un continuum entre ces deux catégories. Un processus peut donc être plus ou moins pré-attentif.

Nous présentons ici quatre des résultats expérimentaux de Treisman qui sont pertinents à nos travaux.

1. Lorsque l'identification de la cible nécessite l'identification d'une seule propriété qui est unique dans l'image, le processus de recherche sera généralement pré-attentif.

Par exemple, chercher une lettre rouge parmi un ensemble de lettres vertes est un processus pré-attentif car, la propriété rouge est présente dans un seul objet. Ici le temps de recherche sera à peu près constant, il n'augmente pas avec le nombre de distractions.

2. Lorsque l'identification de la cible nécessite l'identification d'une *conjonction* de propriétés qui est unique dans l'image, le processus de recherche sera plutôt attentif.

Par exemple, la recherche d'un X de couleur rouge parmi des X de couleur vert sera une tâche pré-attentive parce que la cible se distingue des distractions par une seule dimension. Mais faire la recherche d'un X de couleur rouge parmi un ensemble de X de couleur vert et de T de couleur rouge nécessitera un processus attentif car l'identification des deux propriétés (rouge et la forme X) en conjonction sera nécessaire pour identifier la cible. La recherche se fera de façon séquentielle, et le temps de recherche sera plus ou moins proportionnel au nombre de distractions dans l'image.

3. Si une cible ne se distingue des distractions que par *l'absence* de propriété pour une dimension, le processus sera fortement attentif. Si la cible ne se distingue que par la *présence* d'une propriété, la processus sera fortement pré-attentif.

Un cercle traversé d'une ligne parmi des cercles simples sera identifié de façon pré-attentive alors qu'un cercle simple parmi des cercles traversés d'une ligne sera identifié de façon attentive.

4. Une cible qui a une propriété plus grande et plus visible que les distractions sera trouvée plus facilement qu'une cible qui a une propriété plus petite que les propriétés des distractions.

Une ligne de grande taille parmi des lignes de petite taille sera identifiée plus facilement qu'une ligne de petite taille parmi des lignes de grande taille.

Dimension séparable, configurale et intégrale

De son coté, Pomerantz [66] explore plus en détails la séparation des dimensions. Certaines dimensions peuvent, de par leur disposition, être plus ou moins faciles à séparer visuellement et peuvent s'intriquer lors d'un processus pré-attentif. Par exemple, les deux dimensions d'un rectangle, la largeur et la hauteur, sont intriqués. La perception de l'une des dimensions sera affectée par la valeur de l'autre dimension.

Pomerantz fait le différence entre trois sortes d'intrication entre les dimensions. Un groupe de dimensions est dit *séparable* si le temps pour distinguer une variation sur une des dimensions est le même que la dimension soit présentée seule ou qu'elle soit présentée avec les autres dimensions. Le groupe de dimensions est dit *configural* si le temps pour faire n'importe quelle distinction entre deux propriétés sur une même dimension varie si la dimension est présenté seule. Finalement, si lorsqu'un groupe de dimensions présente une corrélation dans leurs variations à travers l'ensemble de glyphs, et que cela entraîne une diminution dans le temps nécessaire pour effectuer une tâche quelconque, on dit de ce groupe de dimensions qu'il est *intégral*.

Utiliser des dimensions configurales pour visualiser des données qui ne présentent pas de corrélation aura donc comme effet d'interférer avec les processus pré-attentifs. Bien qu'il soit possible d'améliorer l'efficacité d'une visualisation en codant le même ensemble de données ou des ensembles de données qui ont une grande corrélation

dans deux ou plusieurs dimensions intégrales, les gains réalisés sont généralement faibles[87].

2.2.2 Dimensions disponibles

Échelle discrète et continue

Treisman décrit une dimension comme étant constituée d'un ensemble de propriétés mutuellement exclusives. Cette définition entraîne la conséquence que seules les valeurs sur une échelle discrète peuvent être codées dans une dimension. Or certaines dimensions peuvent aussi être utilisées pour coder des échelles continues. La taille d'un glyph, par exemple, peut varier de façon continue d'un minimum à un maximum et la couleur d'un glyph peut varier de façon continue de blanc à noir en passant par tous les tons de gris. Cependant, les processus pré-attentifs décrits à la section précédente ne pourront se produire que pour des valeurs suffisamment différentes pour pouvoir être considérées mutuellement exclusives.

Limitations

On appelle *longueur d'une dimension* le nombre de propriétés différentes que la dimension peut prendre tout en gardant possible leur distinction par un processus pré-attentif. La longueur de la dimension constitue donc une limite quant au nombre de données différentes qu'elle peut coder tout en conservant la possibilité que des processus pré-attentifs se produisent. Par exemple, on verra plus loin que la dimension de la couleur ne peut supporter que 7 ou 8 propriétés différentes.

Dans le cas des dimensions qui varient de façon continue, la limite est d'une autre nature. Si on utilise la dimension pour des données discrètes, on s'intéressera au nombre maximum de propriétés qui peuvent être distinguées de façon pré-attentive. Mais si on utilise la dimension pour des données continues, le concept de longueur ne s'applique pas car la dimension contient une infinité de propriétés. Dans ce cas un processus de détection sera pré-attentif seulement si les propriétés sont suffisamment éloignées.

Les dimensions disponibles

Un grand nombre d'études ont été faites pour identifier les dimensions qui permettent les processus pré-attentifs de se produire. Malheureusement, la littérature est souvent contradictoire sur ce sujet. Le Tableau 4 énumère les variables visuelles les plus communes, le nombre de dimensions qui peut être incorporé, ainsi que la longueur de ces dimensions. Les compromis nécessaires pour pallier aux contradictions qu'on retrouve dans la littérature ont été faits. La plupart de ces dimensions peuvent se comporter comme des dimensions discrètes ou continues. Lorsqu'un point d'interrogation apparaît comme valeur dans la dernière colonne, cela signifie que la publication ne spécifie pas explicitement de longueur pour la variable visuelle en question.

Variable Visuelle	Nombre de dimensions supporté	Longueur des dimensions (discrète)
Position	2 [7] 3 [87]	Limité par la surface disponible Limité par la surface disponible
Taille	1 [7] 6 [87]	20 4
Forme	1 [7] [80]	Infinie
Couleur	1 [40] 1 [87] 3 [85][86]	7 8 ?
Valeur	1 [7]	7
Texture	1 3 [87] 3 [7]	Très grande ? ?
Courbure	1 [80]	3
Orientation	1 [80] [7] 3 [87]	4 4

Tableau 4: Les caractéristiques des variables visuelles selon plusieurs auteurs (Ware, Treisman, Bertin et Healey)

Position. Tout d'abord, on peut coder deux dimensions de par la position du glyph dans le plan si ces deux dimensions sont disponibles. Selon Ware, on peut même coder trois dimensions en créant un effet de profondeur. La longueur de ces dimensions n'est limitée que par la taille de l'espace disponible pour la visualisation.

Taille. Utiliser la taille pour coder des dimensions consiste à faire varier la dimension du glyph d'une façon ou d'une autre. On peut changer le rayon d'un cercle (une

dimension), et on peut changer la largeur et la hauteur d'un rectangle (deux dimensions). Cependant, pour Bertin, seule une homothétie est considérée comme un changement de taille. Changer la largeur et la hauteur par différents ratios est considéré comme un changement de forme. Complètement à l'opposé, Ware suggère qu'on peut coder jusqu'à six dimensions de longueur quatre avec la taille d'un glyph en forme d'étoile (voir Figure 2) !.

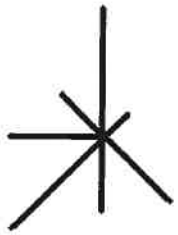


Figure 2: Six dimensions codés dans la taille d'un seul glyph, une dimension par branche.

Dans le cas du rectangle et encore plus dans la cas de cette étoile, les dimensions sont intégrales et configurales. Comme mentionné précédemment, cela peut constituer un avantage ou un désavantage.

Forme. Théoriquement, il existe un nombre infini de formes différentes. La forme d'un glyph peut donc fournir une dimension d'une très grande longueur.

Couleur. Treisman identifie la couleur comme étant l'une des variables visuelles qui se prête le mieux aux processus de détection pré-attentif. Les auteurs sont tous en désaccord en ce qui concerne le nombre de dimensions dans la couleur et leur longueur. Bertin sépare cette variable en deux dimensions : la couleur dont les propriétés sont sur l'axe des couleurs pures de violet à rouge, et la *valeur* (un bel exemple de terme mal choisi qui abondent dans l'ouvrage de Bertin[7]) dont les propriétés sont, en gros, les tons de gris.

Ware[85], lui, prétend pouvoir coder 3 dimensions continues dans la couleur avec une efficacité égale aux trois dimensions de position. Mais dans le même ouvrage il avoue qu'au plus 8 couleurs différentes peuvent être identifiées pré-attentivement. Il avoue

également que si 3 dimensions sont codées avec la couleur, elles seront fortement configurales.

Pour cette variable, l'avis le plus convaincant nous vient de Healey et al.[39] Ces auteurs tentent de définir un ensemble optimal de couleurs discrètes pour une seule dimension. Ils exigent que toute couleur puisse être détectée pré-attentivement et ce, même en présence de toutes les autres. De plus, ils exigent que toutes les couleurs soient aussi faciles à distinguer les unes des autres. À la suite de plusieurs expériences, il en viennent à la conclusion qu'un maximum de sept couleurs peuvent être utilisées simultanément tout en satisfaisant les conditions ci-haut mentionnées. Ces sept couleurs forment donc des candidates parfaites pour la représentation de données qui sont sur une échelle nominale.

En général, si l'on veut obtenir des résultats pré-attentifs, il est plus prudent de ne représenter qu'une seule dimension avec la couleur. Si on a affaire à des données discrètes, on obtient un résultat optimal en utilisant les sept couleurs de Healey et al. Si plus de sept propriétés sont nécessaires, on peut utiliser plus de sept couleurs mais en les gardant aussi loin dans l'espace de couleur que possible[39][40].

Pour des données sur une échelle continue, il convient de choisir un axe dans l'espace des couleurs comme l'axe des tons de gris comme le suggère Bertin.

Texture. Selon Ware, pas moins de trois dimensions peuvent être incorporées dans une texture soit l'orientation, la taille et le contraste. Healey est du même avis, mais présente trois dimensions différentes soit la densité, la régularité et la hauteur. Aucun de ces deux auteurs ne se fait vraiment précis quant à la longueur de chacune de ces trois dimensions.

Un peu comme dans le cas de la forme, on peut utiliser la texture pour coder une seule dimension avec une très grande longueur puisqu'il existe un très grand nombre de texture différentes.

Courbure. La courbure d'une ligne est une dimension appropriée pour les processus pré-attentifs. Les expériences de Treisman démontrent que lorsque seulement deux

propriétés très différentes (ligne droite/demi-cercle) sont présentes, le processus de détection est très pré-attentif, mais il semble évident que trois propriétés au moins peuvent être distinguées de façon pré-attentive [82].

Orientation. La plupart des auteurs s'entendent pour dire que l'orientation d'une ligne peut fournir une dimension. Bertin propose une limite de quatre pour le nombre maximum d'orientations qui sont détectables pré-attentivement.

Autres. Plusieurs autres variables existent ou peuvent être inventées (voir [87] page 166 et [82]. L'ensemble des variables présenté dans cette section n'est nullement exhaustif mais il nous sera suffisant pour notre propos.

2.2.3 Visualisation d'information

Les progrès récents au niveau du matériel informatique et des logiciels ont rendu possible le stockage d'une quantité phénoménale de données, et il devient de plus en plus difficile d'explorer et de faire l'analyse de ces données. C'est pourquoi récemment, le domaine de la visualisation d'information a reçu énormément d'attention. La problématique s'applique au génie logiciel car les logiciels de grandes tailles peuvent contenir plusieurs centaines voir plusieurs milliers de classes. La quantité de données à traiter pour effectuer des tâches d'exploration, de compréhension et d'évaluation est donc énorme. La visualisation est une technique appropriée ici car elle permet de considérer des quantités énormes de données en parallèle, à condition d'avoir une technique de visualisation adaptée à l'ensemble des données.

La méthode de Bertin

L'ouvrage classique *Sémiologie Graphique* [7] du cartographe Jacques Bertin, fait beaucoup plus que de suggérer des dizaines, voire des centaines de façon de visualiser des données à plusieurs dimensions, et il propose une méthode pour construire les images en se basant sur les propriétés de 6 *variables rétiniennes* (forme, orientation, couleur, grain, valeur, taille) du coté de la visualisation, et sur les propriétés des ensembles de données à visualiser. Encore plus intéressant, Bertin propose une classification systématique de l'usage des variables visuelles par rapport au type de

données, au nombre de dimensions et à la longueur des dimensions. Bertin introduit le concept d'*efficacité* d'une image, qui est sa capacité à communiquer l'information dans le plus court laps de temps possible. La méthode qu'il propose vise à maximiser l'efficacité des images. Malgré son âge avancé, Sémiologie Graphique est une source inépuisable de nouvelles idées pour inventer des nouvelles techniques de visualisation.

Le Tableau 5 résume les propriétés des variables visuelles de Bertin. Ce tableau permet de choisir une variable visuelle appropriée selon les caractéristiques de l'ensemble de données à visualiser. Bien que les variables visuelles de Bertin soient différentes des variables visuelles définies dans le Tableau 4, les correspondances sont faciles à faire.

Variables visuelles	Niveau des variables visuelles			
	Association	Sélection	Ordre	Quantité
Variables du plan x et y	X	X	X	X
Taille		X	X	X
Valeur		X	X	
Grain (Texture)	X	X	X	
Couleur	X	X		
Orientation	X	X		
Forme	X			

Tableau 5 : Propriétés des variables rétiniennes de Bertin (reproduit de [7] p.96)

Ce tableau nous permet de juger de l'aptitude des variables visuelles pour différentes tâches reliées à la visualisation. Une variable *associative* est une variable visuelle dont les propriétés ont une visibilité uniforme, et dont aucune d'entre elles n'attire l'attention plus que les autres. La taille n'est pas une variable associative car les objets graphiques de plus grandes tailles sont plus visibles. Une variable est dite *sélective* si elle permet de repérer facilement et rapidement tous les glyphs ayant une même propriété. Étonnamment, Bertin considère que la seule variable qui empêche cela de se produire c'est la forme. Une variable est dite *ordonnée* si les différentes valeurs ont un ordre naturel et universel. Finalement, Bertin identifie les variables du plan et la taille comme étant des variables quantitatives.

La deuxième partie de l'ouvrage de Bertin est consacrée à proposer une visualisation efficace pour un nombre impressionnant de combinaisons de variables visuelles, de type de visualisation et de longueur de variables. Les principes par lesquels ces visualisations sont proposées reposent complètement sur l'expérience et l'intuition de l'auteur. Aucun résultat d'expérience, aucune théorie, ne semble avoir été utilisé (l'ouvrage ne contient aucune référence !). Les visualisations proposées n'en sont pas moins impressionnantes de par leur efficacité et leur ingéniosité.

Classification des techniques de visualisation

En visualisation d'information, le vocabulaire des modèles entité-relation est souvent employé pour décrire les données [87]. Les objets à visualiser sont appelés des *entités,* et il existe des *relations* entre ces entités. De plus, les entités et les relations peuvent avoir des *attributs*. Les *attributs* sont simplement des données sur une certaine échelle. Ici, la visualisation d'information rejoint la théorie de la mesure car on peut utiliser les types d'échelles les plus communs soit nominal, ordinal, intervalle, ratio et absolu. Bertin, lui, ne fait pas ce rapprochement de façon aussi explicite.

Plusieurs techniques de visualisation sont apparues récemment, et quelques propositions ont été faites pour classer ces techniques. Keim propose un ensemble de quatre critères pour classifier les différentes techniques de visualisation : le type de données à visualiser, la technique de visualisation elle-même, la technique d'interaction utilisée et la technique de distorsion utilisée [47]. Des exemples pour chacun des quatre critères sont donnés dans le Tableau 6.

Critères de classification des techniques de visualisation	
Critère	**Exemple**
Type d'ensemble de données	Unidimensionnel, bidimensionnel, multidimensionnel, les graphes et les hiérarchies, algorithmes et logiciels, textes et hypertextes
Technique de visualisation elle-même	Technique standard en 2D/3D, technique orienté pixel, technique orientés icône, technique de transformation géométrique.
Technique d'interaction	Filtrage, zoom, liens hypertexte
Technique de distorsion	Hyperboliques, sphériques

Tableau 6: Critères de classification des techniques de visualisation selon Keim [47]

Les ensembles de données de type uni-, bi-, et multidimensionnel sont, d'après le vocabulaire des modèles entité-relation, un ensemble d'entités avec un, deux ou plusieurs attributs mais sans relation entre les entités. Cette simplicité laisse énormément de flexibilité pour la visualisation car les deux dimensions du plan ne sont pas occupés.

Pour visualiser un ensemble de données unidimensionnel, on a souvent recours aux diagrammes statistiques les plus communs. L'histogramme et le graphique en tarte sont particulièrement bien adaptés pour les données sur une échelle de type nominal ou ordinal alors que le graphique en bâton et la courbe de densité sont bien adaptés pour des données sur une échelle de type quantitatif (intervalle, ratio, absolue). Pour visualiser un ensemble de données bidimensionnel, on utilisera très souvent un graphique à point si les deux dimensions sont sur une échelle de type quantitatif. Si l'une des dimensions est sur une échelle nominal ou ordinal, le graphe à barre convient. Lorsqu'un ensemble de données a plus de 3 dimensions, on dit qu'il est multidimensionnel. Ces ensembles sont problématiques car la visualisation se fait le plus souvent sur une surface physique en deux dimensions comme un écran d'ordinateur ou une feuille de papier. Dans les dernières années, un grand nombre de nouvelles techniques de visualisation ont été proposées pour résoudre ce problème [16].

D'autres ensembles de données comprennent des relations entre les entités. Souvent, exprimer ces relations graphiquement nécessitera l'utilisation des deux dimensions du

plan qui pouvaient auparavant être utilisés pour des attributs. Par exemple, si la relation impose un ordre hiérarchique sur les entités, comme c'est le cas pour l'héritage d'un système orienté objet, l'arbre sera souvent la technique utilisée. La position du nœud dans le plan sera donc déterminée par la relation hiérarchique. Les attributs des entités devront être visualisés autrement qu'avec la position du nœud dans le plan. C'est du domaine de la visualisation d'information de trouver des façons d'accomplir cela.

La *technique d'interaction* décrit la façon par laquelle l'utilisateur fait des changements à la visualisation en la manipulant directement et dynamiquement. Elle décrit aussi comment différentes visualisations peuvent interagir. Les techniques de distorsion permettent de voir instantanément et dynamiquement une partie de la visualisation avec un plus grand niveau de détails tout en conservant une vue d'ensemble des données[47].

2.3 Les cadres d'applications orientés objet

Le but premier des cadres d'applications est la réutilisation de solutions au niveau de l'architecture des logiciels. Un cadre d'applications fournit certaines fonctionnalités centrales qui sont communes à une classe d'applications. Il utilise les concepts orientés objet fondamentaux comme l'abstraction, le polymorphisme et l'héritage pour permettre au programmeur d'ajouter les fonctionnalités spécifiques à ses besoins. Un cadre d'applications est en fait une application semi-complète qu'il faut compléter pour obtenir les fonctionnalités souhaitées. Les cadres d'applications se distinguent des librairies simples de par plusieurs aspects [30] :

- Inversion du flot de contrôle

Contrairement aux librairies logicielles où le programmeur est celui qui appelle les fonctionnalités au moment désiré, les cadres d'applications s'occupent généralement eux-mêmes du flot de contrôle.

- Architecture générale du système ou du sous-système

Dans un système orienté objet, le cadre d'applications occupe généralement le haut de la hiérarchie d'héritage. Il définit les interfaces des différentes parties du système ou du sous-système et comment elles communiquent.

- Classes abstraites et interfaces

Un cadre d'applications est généralement constitué de classes abstraites que le programmeur doit sous-classer et d'interfaces auxquels le programmeur doit se conformer.

Les cadres d'applications ont été premièrement utilisés pour les interfaces graphiques mais sont maintenant utilisés dans plusieurs domaines [31]. Typiquement, un cadre d'applications pour interfaces usagers s'occupe de créer, de dessiner et d'animer les différentes composantes graphiques (bouton, ascenseur, menu). Il s'occupe également de transformer les actions de l'utilisateur (mouvement de souris, pression de bouton), en appel vers les fonctionnalités que le programmeur aura implantées et qui sont propres à l'application. Ici on remarque l'inversion de contrôle : c'est le cadre d'applications qui, à la demande de l'utilisateur, appelle les fonctionnalités propres à l'application. C'est également le cadre d'applications qui dicte comment les différentes parties du logiciel, la vue, le modèle, et les contrôles communiquent (voir patron MVC dans [15]).

Une même application peut faire usage de plus d'un cadre d'applications pour obtenir différents types de fonctionnalités. Par exemple, une application qui permet de dessiner peut à la fois faire usage d'un cadre d'applications pour ses interfaces graphiques usagers conventionnelles (bouton, menu) et un autre cadre d'applications pour les fonctionnalités plus avancées liées au dessin.

Le but de ce travail est de fournir des solutions logicielles réutilisables à plusieurs des problèmes présentés au Chapitre 4. Implanter ces solutions sous la forme d'un cadre d'applications les rendra facilement réutilisable dans d'autres applications.

2.4 Modèle et méta-modèle

2.4.1 UML

Dans les dernières années, le Unified Modelling Language [63] (UML ci-après) a été accepté comme le standard pour la modélisation des systèmes orientés objet. UML est un langage graphique permettant de spécifier, de visualiser, de construire et de documenter différents aspects d'un système logiciel. Le langage UML peut être divisé en deux grandes parties: le méta-modèle et les diagrammes.

Les principaux diagrammes sont au nombre de neuf. Ces neuf diagrammes présentent une vue abstraite de différents aspects du système. La documentation UML fournit une description assez précise de la notation graphique à utiliser pour la création de ces diagrammes.

La sémantique des éléments présents dans ces diagrammes est décrite à l'aide du méta-modèle UML. Le méta-modèle UML se présente comme une hiérarchie de classes distribuée dans plusieurs packages avec une description en langage naturel pour chaque éléments (classe et package) et quelques contraintes spécifiées en langage OCL (Object Constraint Language) [42]. Chaque classe de cette hiérarchie sert à décrire la sémantique d'un type d'élément graphique que l'on retrouve dans un ou plusieurs diagrammes. Les éléments du méta-modèle sont appelés méta-éléments.

Une caractéristique importante de UML est son extensibilité. En effet, UML possède trois mécanismes d'extension définis à même le méta-modèle qui permettent de définir de nouveaux méta-éléments adaptés à des tâches spécifiques.

Les trois mécanismes d'extension sont la classification, les propriétés et les contraintes. La classification se fait à l'aide du méta-élément *Stereotype*. Elle permet d'étiqueter un élément pour signaler son appartenance à un groupe d'éléments ayant une sémantique distincte. Le deuxième mécanisme d'extension, les propriétés, réalisées à l'aide des méta-éléments *TaggedValue* et *TagDefinition,* permet d'associer une ou plusieurs propriétés supplémentaires à un méta-élément. Finalement, les contraintes, réalisées à l'aide du méta-élément *Constraint* permettent d'associer une

ou plusieurs nouvelles contraintes à un méta-élément. Ces contraintes peuvent être écrites en langage OCL ou en tout autre langage qui convient.

Un ensemble cohérent de ces méta-éléments constitue un profil UML. Par exemple, le profil RCR (Rétroconception, Compréhension, Réingénierie) [77], définit 52 nouveaux méta-éléments permettant la modélisation détaillée de logiciels conçus dans des langages impératifs classiques ou orientés objet. Les entités modélisées par les méta-éléments du profil RCR comprennent les blocs, les énoncés, les expressions, les variables et les opérateurs primitifs qui constituent les expressions complexes.

2.4.2 MOF

La plupart des outils de modélisation utilisent maintenant les diagrammes d'UML [77]. Cette standardisation a permis une certaine interopérabilité entre ces outils. Pour encourager cette interopérabilité, l'*Object Management Group* (OMG ci-après) qui est l'organisation responsable de l'évolution d'UML s'est dotée du *Meta Object Facility* (MOF ci-après)[62], une architecture de méta-modélisation à quatre niveaux comme illustrée à la Figure 3.

Le MOF définit un méta-métamodèle qui se veut capable de définir la sémantique de n'importe quel méta-modèle imaginable, y compris lui-même. Sa définition est donc circulaire. Par conséquent, MOF peut théoriquement être utilisé pour décrire n'importe quel méta-modèle basé sur les concepts orientés objet. Le méta-modèle MOF se situe au niveau M3 sur la Figure 3 et sert à définir des méta-modèles de niveau M2 comme le méta-modèle UML. Au niveau M1, on retrouve des modèles qui sont définis par le méta-modèle de niveau M2. Dans le cas d'UML, on retrouve des instances des neuf diagrammes qui sont définis à l'aide du méta-modèle UML. Au niveau M0 on retrouve les données concrètes qui sont modélisées. Dans le cas de la modélisation de système avec UML, ces données sont, par exemple, des objets concrets, c'est-à-dire des instances de classes du système.

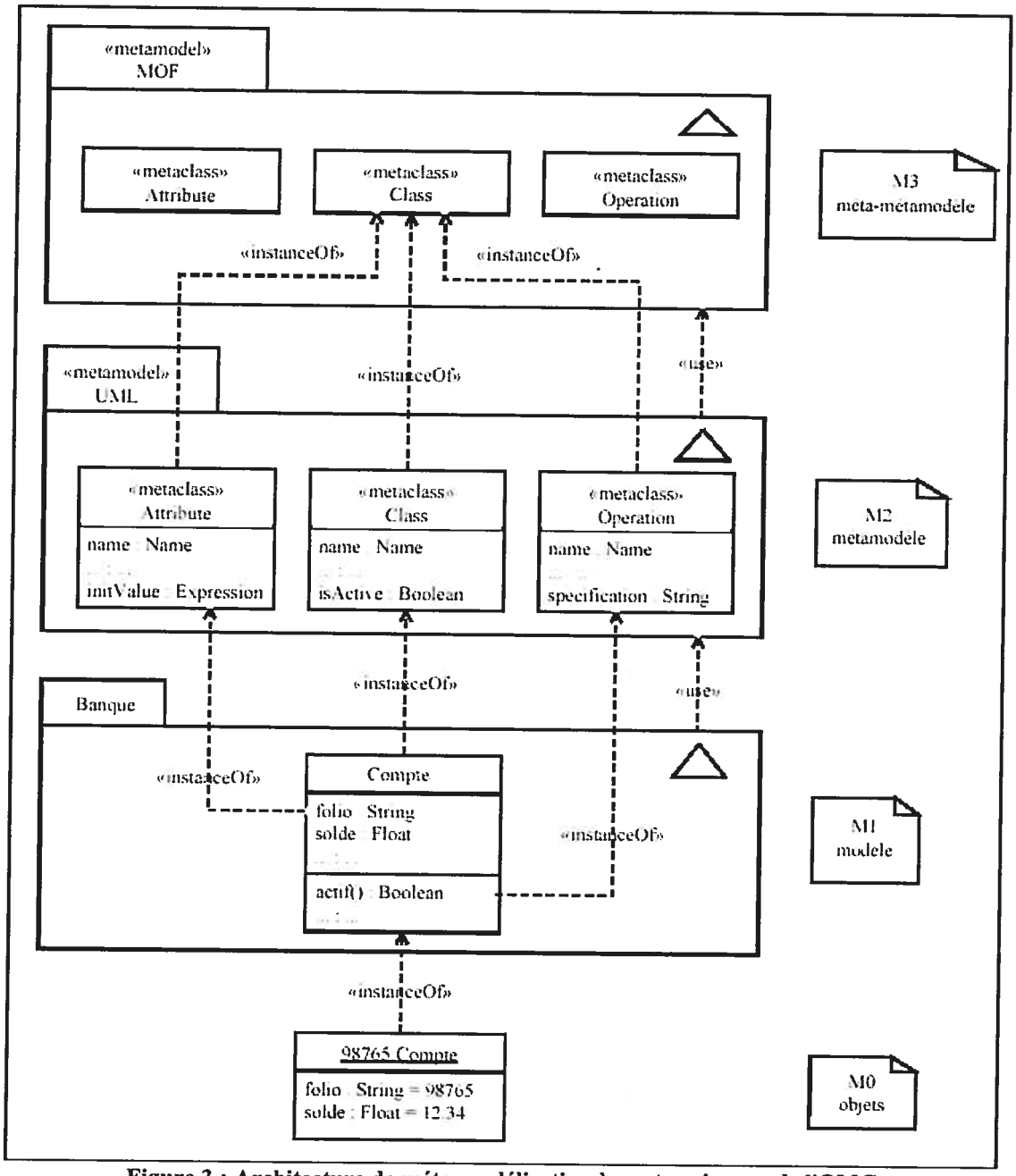


Figure 3 : Architecture de méta-modélisation à quatre niveaux de l'OMG (reproduit de [77])

2.4.3 OCL

Dans un modèle, il est parfois pratique de définir des contraintes sur ses éléments pour assurer la bonne forme du modèle. L'*Object Contraint Language* (OCL ci-après) (voir [63] chapitre 6] et [42]) est un langage formel qui permet de faire cela. Typiquement, on utilise OCL pour spécifier des conditions nécessaires pour qu'une

instance d'un modèle ou d'un méta-modèle soit bien formé. OCL permet donc d'enrichir la syntaxe des modèles et des méta-modèles.

OCL est utilisé à tous les niveaux de l'architecture MOF. On utilise OCL dans le méta-métamodèle MOF pour dicter des conditions sur la formation des méta-modèles comme le méta-modèle UML. Dans ce dernier, OCL est utilisé pour dicter des conditions sur la formation des modèles, c'est à dire sur la formation des neuf diagrammes UML. Finalement, on peut utiliser OCL à même les diagrammes UML pour spécifier des conditions sur les objets qui seront créés à l'exécution du programme.

OCL est un langage à expressions, c'est-à-dire qu'une expression OCL ne peut pas changer l'état d'un système sur lequel elle s'applique. OCL permet cependant de faire n'importe quel calcul sur les éléments d'un modèle. Le langage fournit, entre autres, un ensemble de types primitifs (ex : Boolean, Integer, Real, String), des opérations applicables sur ces types (ex : +, - , / , * , abs( )) et des types d'ensembles (ex : Collection, Set, Bag, Sequence) accompagnés d'opérations pour les parcourir.

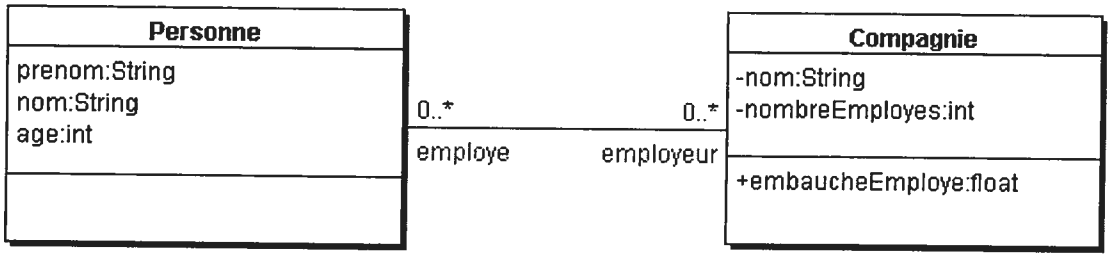


Figure 4: La classe Compagnie

Dans le langage OCL, on retrouve trois types de contraintes : les invariants, les pré-conditions et les post-conditions. Des exemples pour chacun de ces trois types de contraintes sont donnés dans les paragraphes suivant en utilisant la Figure 4. Ces exemples ne donnent qu'un bref aperçu des possibilité de OCL.

Les invariants

Les invariants sont des expressions booléennes qui sont associées avec un élément du modèle et qui doivent être vraies en tout temps pour que le modèle soit intègre. Par exemple, avec OCL, pour les classes *Compagnie* et *Personne* de la Figure 4, on peut

spécifier que la compagnie doit avoir plus de 50 employés avec l'instruction suivante :

```
context Compagnie inv:
        self.nombreEmployee > 50
```

Ici, *self* désigne une instance de la classe *Compagnie*. OCL possède également plusieurs opérateurs d'ensemble qui permettent de traverser un ensemble sans avoir à définir de boucle explicitement. Avec l'instruction suivante on peut vérifier que tous les employées sont âgés d'au moins 15 ans :

```
Context Compagnie inv :
        self.employe -> forAll (p: Personne | p.age ≥ 15)
```

Ici, *forAll* est un opérateur d'ensemble qui signifie que la condition (*age ≥ 15*) entre parenthèses doit être vraie pour tous les employés associés avec une instance de la classe *Compagnie* si on veut que l'invariant soit satisfait.

Les pré-conditions et les post-conditions

Les pré-conditions sont généralement associées à des opérations ou à des méthodes. Une pré-condition est une expression booléenne qui impose des conditions sur les paramètres de la méthode ou de l'opération. L'instruction suivante vérifie qu'une personne a plus de 15 ans avant de l'embaucher :

```
Context Compagnie :: embaucheEmploye(Personne p) : Real
  pre : p.age ≥ 15
```

Les post-conditions, elles, sont utilisées pour poser des conditions sur la valeur retournée par une méthode ou une opération. La méthode *embaucheEmploye* par exemple renvoie un nombre réel qui est le salaire du nouvel employé. Avec une post-condition on peut vérifier que ce salaire n'est pas supérieur à 50 000 $.

```
Context Compagnie :: embaucheEmploye(Personne p) : Real
  post : result < 50 000.0
```

Chapitre 3 : État de l'art

Il existe déjà une grande quantité d'outils académiques et commerciaux qui permettent de faire le calcul de métriques et/ou qui utilisent des techniques de visualisation. Ce chapitre décrit l'état de l'art dans ce domaine en examinant les caractéristiques de quelques outils représentatifs.

3.1 La visualisation logicielle

La visualisation est maintenant présente dans quasiment tous les outils CASE, tant du coté commercial que du coté académique, et ces outils servent à tous les niveaux du cycle de vie du logiciel de la conception initiale à la maintenance. Certains outils comme ceux basés sur UML [88][80][67] aident à la modélisation. D'autres outils assistent les programmeurs en permettant de visualiser des choses comme les algorithmes, le code source et la structure du logiciel.

Une étude quantitative comparative de plusieurs outils de visualisation logicielle est récemment parue [5][6].

3.2 Librairies permettant la visualisation de données

D'un autre coté, le marché des composantes logicielles offre plusieurs dizaines de librairies permettant de construire toute sorte de visualisation pour des ensembles de données arbitraires.

La plupart de ces librairies présentent des visualisations en deux dimensions et en trois dimensions. Elles offrent en général les graphiques statistiques les plus communs comme le graphique en tarte, le graphique à barre, le graphique à point, l'histogramme et graphique à lignes brisées mais aussi plusieurs nouvelles techniques

de visualisation très puissantes. Certaines de ces librairies offrent des techniques de visualisation dynamiques et interactives où l'utilisateur peut manipuler directement ce qu'il voit [45][64], le but de ces manipulations étant de révéler ou cacher de l'information, de faire un zoom ou de filtrer de l'information. Finalement, certaines librairies ne sont disponibles qu'en tant que service web [1].

3.3 Outils pour le calcul et la visualisation des métriques

La plupart des outils académiques et commerciaux qui permettent le calcul des métriques fournissent également une façon de visualiser les résultats. Au niveau académique cependant, les phases de calcul et de visualisation se font souvent complètement séparément. C'est le cas pour une étude réalisée par Eick et al. [27] sur les requêtes de changement des logiciels. Les auteurs calculent dans un premier temps les métriques en se servant d'une énorme base de données CVS, pour ensuite utiliser un outil de visualisation commercial qui n'est en rien conçu spécifiquement pour la visualisation de métriques.

En général les outils commerciaux permettent de calculer un ensemble pré-déterminé de métriques et proposent quelques techniques de visualisation simple ainsi qu'un engin de génération de rapport en plusieurs formats. Le Tableau 7 présente quelques outils commerciaux et académiques représentatifs des outils pouvant être utilisés faire la visualisation des métriques.

Outils commerciaux	Web Gain Studio [88]
	Krakatau Metrics Professional [50]
	Discover [25]
Outils académiques	Crocodile [74] [55]
	Code Crawler[51] [52] [53][54]

Tableau 7: Outils commerciaux et académiques évalués

Bien sûr cet ensemble d'outils n'est pas exhaustif, il représente bien les outils qui sont présentement disponibles.

3.3.1 Outils commerciaux

WebGain Studio

WebGain Studio, est un environnement de développement pour Java. Il fournit un ensemble d'outils qui aide à la création d'applications client-serveur complexes basés sur les standards J2EE [44]. Il supporte la génération automatique de code et la synchronisation du code source à partir de diagrammes UML. Il permet également le déploiement automatique d'applications vers le serveur et la création rapide d'interfaces graphiques et de pages web.

L'une de ses composantes, *Quality Analyzer*, procure trois sous-composantes pour surveiller, améliorer et évaluer la qualité des systèmes Java. Pour évaluer la qualité des systèmes Java, treize métriques sont proposées. Ce sont des métriques connues comme la complexité cyclomatique, les métriques de Chidamber et Kemerer, et quelques métriques de taille. Pour visualiser ces métriques, l'utilisateur a trois choix. Premièrement, il peut utiliser une feuille de calcul interactive qui permet de parcourir les entités du système hiérarchiquement. Un graphe de Kiviat ainsi qu'un graphe à barres sont également disponibles. Pour chacune des métriques, l'utilisateur peut spécifier un maximum au-delà duquel les résultats des métriques apparaissent en rouge dans la feuille de calcul. Dans cet outil, les données semblent être calculées directement à partir du code source sans représentation intermédiaire du code source.

Krakatau Metrics Professional

Krakatau Metrics Professionnal est un outil dédié spécifiquement à l'évaluation quantitative des logiciels Java et C++. L'outil permet entre autres de calculer les métriques de Chidamber et Kemerer, les métriques de Halstead, les métriques du projet MOOD[60] ainsi que des métriques de complexité et de taille. Il est également possible de mesurer la différence entre deux versions d'un même logiciel avec des métriques de changement. En tout, un ensemble de plus de 70 métriques sont proposées. Des ensembles de métriques différents sont proposés pour les systèmes Java et C++.

L'outil fournit une interface graphique et une interface ligne de commande avec les mêmes fonctionnalités. L'utilisateur peut visualiser les métriques de différentes façons. Premièrement, l'utilisateur peut consulter les résultats numériques des métriques de son choix dans une feuille de calcul. Il peut aussi obtenir les résultats des métriques dans un graphique à barres qui permet de comparer les résultats d'une même métrique pour plusieurs éléments du système, ou dans un graphique de Kiviat qui permet de visualiser les résultats d'un petit nombre de métriques pour un même élément du système. L'utilisateur peut également définir un intervalle à l'intérieur duquel les résultats des métriques doivent se trouver. Les résultats à l'extérieur de cet intervalle sont affichés d'une couleur différente dans la feuille de calcul et dans l'histogramme, et ils sont facilement identifiables dans le graphique de Kiviat. Finalement, l'utilisateur peut générer un rapport en format HTML ou CSV contenant les résultats désirés.

Discover

L'environnement *Discover*, est composé d'un ensemble d'outils assistant les différentes personnes impliquées dans le développement d'un système logiciel. L'un de ces outils, *Discover Quality*, permet de faire une évaluation quantitative et qualitative des logiciels écrits en Java ou en C/C++. L'outil applique un ensemble prédéfini de 17 métriques au système. Les métriques proposées s'appliquent sur trois types d'éléments, les fichiers, les fonctions (méthodes), et les classes. La plupart des métriques proposées mesurent la présence de défaut à un niveau très bas. Par exemple, une des métriques permet de calculer le nombre d'instructions *switch* qui possèdent une clause *default*. Un petit nombre de métriques mesurent des attributs plus conventionnels comme la taille et la complexité du système. Les résultats des métriques sont intégrés à l'outil principal de navigation de l'environnement. La visualisation des métriques se fait donc à l'intérieur d'une liste, où les résultats numériques sont placés à coté du nom de chaque élément.

Au centre de l'environnement Discover se trouve une base de données qui contient tous les détails du code source. Les résultats des métriques sont partie intégrante de cette base de données. Ils sont calculés lors de la création du dépôt. Un langage de

scriptage basé sur TCL permet de faire des requêtes sur la base de données et ainsi calculer d'autres métriques. Cependant, ces métriques ne peuvent être facilement intégrées dans les outils au même titre que les autres métriques.

3.3.2 Outils académiques

Crocodile

Crocodile est un plug-in pour l'environnement de développement SNiFF[75] développé par le groupe de recherche en génie logiciel de l'Université de Cottbus. Crocodile utilise la table de symbole présente dans SNiFF pour faire le calcul de plusieurs dizaines de métriques. Les métriques sont calculées sur plusieurs types d'éléments, les attributs, les méthodes, les classes, les fichiers et les packages.

Une propriété intéressante de cet outil est le support pour les modèle de qualité définie par décomposition. Il est possible de définir un ensemble d'attributs externes que l'on veut mesurer. Pour chacun de ces attributs on définit un ou plusieurs attributs internes ainsi que les métriques nécessaires pour le calculer. Le modèle de qualité peut être visualisé dans un arbre horizontal avec au bout de chaque branche, un intervalle dans lequel le résultat de la métrique doit se trouver pour être considéré correct. Pour la visualisation des métriques elle-même, elle peut être faite à l'aide d'un graphique à barres ou à l'aide de rapport en format HTML.

Avec Crocodile, il est également possible de définir de nouvelles métriques soit par composition d'autres métriques à l'aide d'opérateurs mathématiques standards, soit à l'aide d'un langage de requête semblable à SQL.

Code Crawler

Code Crawler est un outil mettant de l'avant une approche basée sur un usage intensif de différentes techniques de visualisation pour faire l'exploration des résultats des métriques.

Comme source d'information, l'outil utilise un dépôt dont le schéma, FAMIX[29], a été développé à l'intérieur d'un groupe de recherche. Précisément, l'approche proposée consiste à enrichir des graphes et des diagrammes représentant des éléments

du système avec les résultats des métriques calculés sur ces éléments. Un arbre d'héritage de classes, par exemple, pourra supporter 3 métriques de classes en laissant varier la hauteur, la largeur et la couleur de chaque nœud du graphe en fonction du résultat de la métrique. Code Crawler propose plusieurs autres visualisations originales de ce genre qui démontrent tout le potentiel de cette approche.

Chapitre 4 : Énoncé de la problématique

La création du cadre d'application décrit dans ce mémoire nous a amené à solutionner plusieurs problèmes de natures très différentes. Ce chapitre vise à décrire en détails les obstacles rencontrés. La section 4.1 décrit les nombreuses formes d'ambiguïté que l'on rencontre typiquement dans le processus de calcul des métriques. La section 4.2 examine le problème qui découle de l'hétérogénéité des composantes de visualisation existantes. La section 4.3 discute brièvement de l'utilisation qui peut être faite de la psychophysique dans le domaine de la visualisation des métriques. Finalement, la section 4.4 discute des avantages à implanter notre solution sous la forme d'un cadre d'application.

4.1 Ambiguïté des concepts reliés aux métriques

L'obstacle majeur à l'épanouissement de la recherche dans le domaine des métriques orientées objet et leur acceptation par l'industrie est sans doute le manque de précision que l'on retrouve dans la définition des métriques et de beaucoup de concepts qui s'y rattachent.

Ces ambiguïtés ont nui à la recherche dans ce domaine car elles rendent la reproduction des résultats impossible. La reproductibilité est pourtant un principe de base de la méthode scientifique. Lorsque des résultats scientifiques sont rapportés, le rapport doit contenir toute l'information nécessaire à la répétition de l'expérience. Ceci est nécessaire pour permettre aux autres chercheurs de vérifier les résultats et pour rendre possible leur réutilisation. Avec toutes les ambiguïtés que l'on retrouve

dans le domaine des métriques, ceci se produit très rarement. Les expériences rapportées sont rarement vérifiées ou reprises par d'autres chercheurs ou par d'autres professionnels. Par exemple, dans l'industrie, les résultats des métriques ne sont considérés comme valides qu'à l'intérieur de l'environnement technique où ils ont été obtenus.

La Figure 5 illustre les ambiguïtés les plus importantes que l'on retrouve aujourd'hui dans les travaux liés aux métriques et à la qualité des logiciels. La discussion qui suit met l'emphase sur les métriques de produit pouvant être calculées à partir du code source.

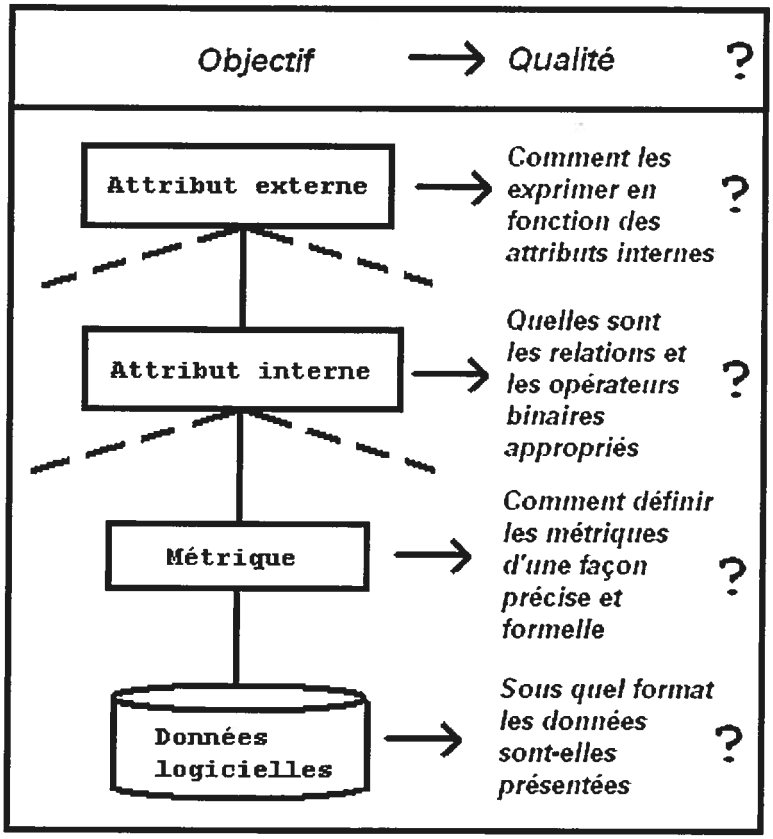


Figure 5: Ambiguïtés dans le domaine des métriques

4.1.1 Format des données logicielles

Un système peut être présenté sous plusieurs formes. La représentation la plus fidèle d'un système est bien sûr le code source du système lui-même. Cependant, calculer les métriques directement depuis le code source n'est pas une stratégie souvent utilisée pour des raisons de performance. Une métrique qui calculerait le nombre de

sous-classes d'une certaine classe, par exemple, aurait besoin de parcourir toutes les classes du système pour découvrir ses sous-classes.

La stratégie la plus souvent utilisée consiste à créer une représentation intermédiaire du système, un modèle, qui est construit à partir d'informations extraites du code source. Ces modèles sont conservés dans un ensemble de fichiers ou dans une base de données pour être ensuite accédés par différents outils, comme les outils qui calculent les métriques par exemple. En général, ces modèles contiennent des informations qui peuvent améliorer la performance de ces outils. Entre autres, la représentation d'une classe contiendra une référence vers chacune de ses sous-classes ce qui évitera d'avoir à parcourir le système au complet pour les retrouver. En revanche, la création de ces modèles peut-être très longue dépendamment du niveau de détail qu'ils contiennent, soit de plusieurs minutes à plusieurs heures pour un système de grande taille.

Tout comme dans le cas de la modélisation de système, la sémantique des éléments de cette représentation intermédiaire, de ce modèle, est souvent exprimée à l'aide d'un méta-modèle (voir section 2.4), et les éléments de la représentation intermédiaire sont considérés comme des instances des méta-éléments du méta-modèle. Ici il faut noter que dans la plupart des méta-modèles, *la sémantique fournie par les méta-éléments ne correspond pas directement à la sémantique des éléments du langage de programmation dans lequel le système étudié est conçu.*

Bien que la représentation intermédiaire à l'aide d'un modèle soit nécessaire, cette nécessité vient avec son lot de problèmes. Deux problèmes sont particulièrement importants :

- Il existe plusieurs méta-modèles différents qui peuvent servir à créer des représentations intermédiaires permettant le calcul de métriques. Par conséquent, une même métrique appliquée sur deux représentations intermédiaires d'un même système instancié à partir de deux méta-modèles différents peut potentiellement donner des résultats différents (voir Figure 6). Un exemple peut être donné avec une métrique très simple et de très haut

niveau soit le *Nombre de classes dans un système* (voir [56] p.95) lorsque appliqué à un système conçu en Java. Une première représentation intermédiaire peut décider de modéliser les concepts de classe et le concept d'interface par deux méta-éléments différents alors qu'une autre peut, raisonnablement, modéliser les interfaces Java comme des classes ayant seulement des méthodes abstraites. La métrique *Nombre de classes dans un système* donnera bien sûr des résultats différents dans les deux cas. Donc, même pour les métriques de design de très haut niveau, la diversité des représentations intermédiaires peut causer des différences dans les résultats des métriques ; pour les métriques de code source de bas niveau portant par exemple sur les appels d'opération, les pointeurs, les références, les possibilités d'erreurs se comptent par dizaine.

- Comme mentionné plus haut, la sémantique fournie par les méta-éléments ne correspond pas toujours directement à la sémantique des éléments du langage de programmation dans lequel le système étudié est conçu. Dans certains cas, cette correspondance n'est tout simplement pas fournie. C'est le cas de UML qui a été conçu pour pouvoir s'adapter à tous les langages de programmation orientés objet. Ceci a pour conséquence que, pour un même langage de programmation, il existe plusieurs façons différentes d'adapter UML. Ces remarques ne s'appliquent pas seulement à la sémantique additionnelle définie par l'utilisateur à l'aide des mécanismes d'extension mais aussi aux concepts sémantiques centraux de UML. Ces différentes adaptations peuvent causer des incohérences au niveau des résultats des métriques (voir Figure 7).

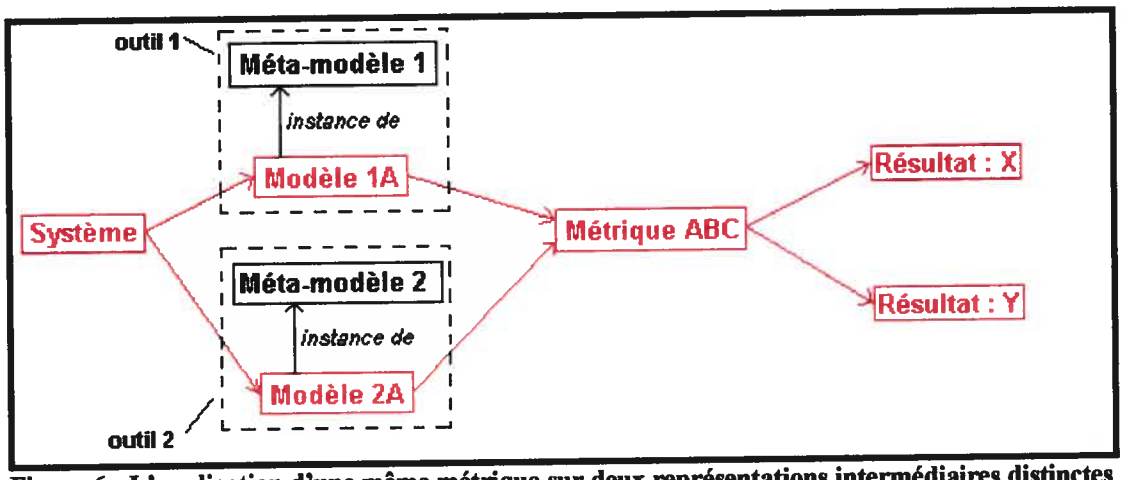


Figure 6 : L'application d'une même métrique sur deux représentations intermédiaires distinctes d'un même système peut donner des résultats distincts

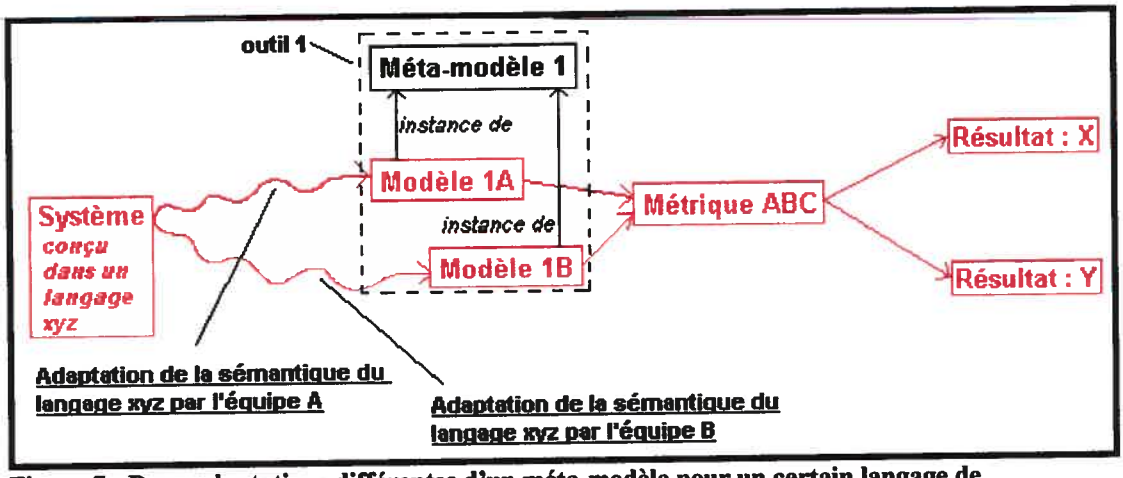


Figure 7 : Deux adaptations différentes d'un méta-modèle pour un certain langage de programmation causera deux représentations intermédiaires distinctes pour un même système et, par conséquent, des résultats distincts pour le calcul des métriques

Il existe une très grande diversité au niveau des méta-modèles utilisés par les outils qui calculent les métriques. En fait, presque chaque outil utilise son propre méta-modèle (voir [77] section 3.3). Certains meta-modèles sont restreints aux composantes de haut niveau (classes, attributs, signature des méthodes, héritage) alors que d'autres permettent de modéliser tous les détails du corps des méthodes. Le premier des problèmes énumérés ci-haut est donc très réel et constitue un obstacle majeur à l'acceptation des métriques.

D'un autre coté, puisque la plupart des outils permettant de calculer des métriques utilisent un méta-modèle qui lui est propre et fournissent un importateur pour chaque langage qu'ils supportent, les occasions d'adapter leur méta-modèle à un nouveau langage de programmation sont rares. Ce problème risque plus de se produire pour

UML, qui tente de s'imposer comme un standard mais qui ne fournit de correspondance précise entre la sémantique fournie par ses méta-éléments et la sémantique des construits des différents langages de programmation. Les concepteurs des différents outils qui utilisent le méta-modèle UML comme format pour la représentation intermédiaire doivent aussi concevoir un importateur pour créer cette représentation à partir du code source. La création d'un tel importateur implique la définition d'une projection entre la sémantique du langage et la sémantique des méta-éléments de UML. Cette projection étant quelque peu laissée à la discrétion des concepteurs, différents importateurs peuvent être conçus pour un même langage, ce qui fait qu'un même système peut se voir représenter de différentes façons même si le méta-modèle utilisé est toujours UML. Par conséquent, si on applique aveuglément une métrique sur deux représentations intermédiaires issues de deux adaptations différentes de UML pour un certain langage, les résultats seront parfois inégaux. Le deuxième problème énuméré ci-haut affecte donc principalement les outils utilisant le méta-modèle UML comme format pour les représentations intermédiaires.

4.1.2 Définition des métriques

Le problème le plus évident du domaine des métriques est au niveau de leur définition. Comme l'explique Briand et al. dans un article sur les métriques de couplage [8], il n'existe pas vraiment de formalisme pour définir les métriques :

> "For example, because there is no formalism for expressing measures, many measures are not fully operationally defined, i.e., there is some ambiguity in their definitions."

Tout comme les métriques de Chidamber et Kemerer, la plupart des métriques orientées objet qui ont été proposées récemment sont principalement décrites en langage naturel et sans explications précises sur la façon de les appliquer dans la réalité. En même temps, ces métriques prétendent être également applicables à tous les langages de programmation orientés objet. Par exemple, dans le premier ouvrage sur le sujet [56], les métriques sont toutes décrites en langage naturel seulement. La situation ne s'est pas beaucoup améliorée depuis.

Leur mise en pratique demande donc un grand nombre de décisions tout à fait subjectives ce qui donne naissance à plusieurs versions différentes de la même métrique. C'est pourquoi il existe plusieurs versions de certaines des métriques de Chidamber et Kemerer comme le démontre des travaux récents[10].

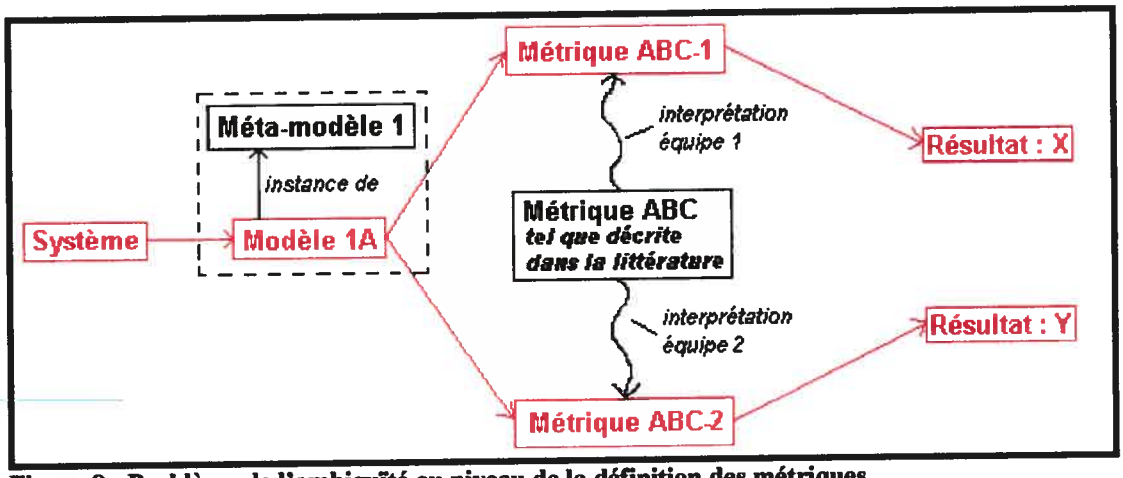


Figure 8 : Problème de l'ambiguïté au niveau de la définition des métriques

Dans la Figure 8 on voit que même si deux équipes de maintenance utilisent la même référence dans la littérature pour utiliser une métrique, et même s'il s'adonne à utiliser le même outil pour donner une implémentation concrète à la métrique, ils doivent l'interpréter en fonction du méta-modèle qu'ils utilisent, ce qui peut causer des incohérences au niveau des résultats de ces métriques.

Ce problème est rendu encore plus aigu par le fait que beaucoup de métriques sont définies comme étant applicables à tous les langages de programmation. Il ne fait aucun doute que des classes écrites en C++, Java et Smalltalk, même si elles réalisent les mêmes fonctionnalités bien définies, sont des objets empiriques bien distincts au sens de la théorie de la mesure. Que l'on tente d'évaluer la complexité, la taille ou tout autre attribut, mesurer l'un ne peut pas être considéré équivalent à mesurer l'autre. De plus, ces différences entre les langages de programmation orientés objet font immédiatement apparaître d'autres problèmes d'interprétation. Par exemple, dans le langage Java on retrouve les concepts de classes internes, de classes imbriquées, de classes anonymes, de classes abstraites et d'interfaces, concepts qui peuvent tous être assimilés ou non au concept de classe. Alors que faire lorsque Chidamber et Kemerer définissent la métrique WMC (*Weighted Method Per Class*) en ne donnant aucune

indication sur ce qu'est précisément une classe? En Java, à quel type de construit cette métrique est-t-elle applicable exactement ?

En fait, les métriques de Chidamber et Kemerer ne sont tout simplement pas des métriques selon la définition donnée dans la section 2.1.2. Car ici, nous sommes bien loin d'une fonction permettant d'associer un nombre réel à chacune des classes d'un système indépendamment de tous autres aspects techniques. Ces métriques sont néanmoins conceptuellement très utiles et peuvent certainement être utilisées pour créer des métriques bien définies.

4.1.3 Relations et opérateurs binaires des attributs internes

Pour qu'une mesure soit considérée comme valide, la théorie de la mesure demande qu'un ensemble de relations et d'opérateurs binaires soient définis et satisfaites lorsque des mesures concrètes sont faites. Or très peu de chercheurs se sont attardés à définir ces relations et ces opérateurs pour les métriques qu'ils proposent, encore moins à vérifier s'ils sont satisfaits. Briand et al. et Zuse sont parmi les rares chercheurs à s'être réellement attardés à ces considérations [9][92]. Ces considérations sont importantes car ce sont ces relations et opérateurs qui déterminent sur quel type d'échelles la mesure de l'attribut se trouvera et par conséquent quelles opérations mathématiques et quel genre de statistiques pourront être appliqués sur les résultats des métriques. De plus, ce sont les caractéristiques de ces relations et opérateurs qui détermineront si une mesure assumant une structure extensive pourra être définie et donc, si une mesure de type ratio pourra être obtenue ou si il faudra se contenter d'une mesure de type ordinal.

4.1.4 Composition des attributs externes

De façon assez surprenante, les modèles de qualité les plus populaires ne fournissent pas de règle pour évaluer les valeurs des attributs externes à partir des attributs internes; un simple lien de dépendance est donné entre eux. C'est le cas pour le modèle de qualité ISO 9126 [43] ainsi que ceux de McCall [59] et de Boehm [12].

4.1.5 La qualité

Finalement, comme mentionné dans la section 2.1.4, la qualité elle-même est un concept ambigu qui dépend largement du point de vue de l'observateur [49]. Avec les métriques et les modèles de qualité, on tente de définir la qualité d'un système en fonction de ses caractéristiques internes mais les caractéristiques internes ne représentent que peu d'intérêt pour l'utilisateur du produit qui, lui, dira que le produit est de qualité s'il fournit les fonctionnalités promises et s'il est fiable.

L'un des objectifs de la collecte de métriques est d'évaluer (ou de prédire) la qualité. Malheureusement, il existe plusieurs points de vue à la qualité et les métriques ne peuvent en présenter qu'un.

4.1.6 Conséquences

De l'avis de Boehm[11], tout comme en cosmologie, les grands progrès théoriques en génie logiciel ne pourront se produire qu'après qu'une grande quantité de données aient été accumulées :

> "the software field cannot hope to have its Kepler or its Newton until it has had its army of Thycho Brahes, carefully preparing the well defined observational data from which a deeper set of scientific insights may be derived".

Cette accumulation de données ne pourra malheureusement pas se produire tant que les ambiguïtés décrites ci-haut n'auront pas été résolues. Pour que les données provenant d'évaluation faites dans différents environnements puissent être accumulées, une façon standard d'exprimer les résultats du calcul des métriques est nécessaire. Certains auteurs ne se font aucune illusion à ce sujet et définissent explicitement les métriques comme n'étant valides qu'à l'intérieur de l'environnement technique et humain dans lesquelles elles ont été définies! Il est vrai que certaines activités liées à la recherche dans le domaine des métriques sont, de façon inhérente, subjectives. La validation de certaines métriques, par exemple, se fait en utilisant les avis des experts ce qui est très subjectif. Cependant, la façon dont les métriques sont définies et calculées n'a pas à être ambiguë. Seul un effort de

standardisation de la part des gens impliqués dans le domaine des métriques est nécessaire. C'est seulement après que ce processus de standardisation aura été accompli que les données provenant de différentes évaluations dans différents environnements pourront être accumulées et comparées et que des progrès théoriques réels pourront être accomplis.

4.2 Interface entre les librairies graphiques et métriques

Bien que la visualisation soit une technique énormément employée en génie logiciel, elle n'est pas beaucoup employée pour la visualisation des métriques. La plupart des outils qui permettent de calculer les métriques ne fournissent qu'un ensemble très restreint de visualisation, la feuille quadrillée et le graphique de Kiviat étant les plus communs. Dans la plupart des cas, il n'est possible ni d'ajouter de nouvelles métriques, ni d'ajouter de nouvelles visualisations. Ceci est très malheureux car tant au niveau de la recherche dans le domaine des métriques que de la recherche dans le domaine de la visualisation d'information, les innovations sont constantes. De nouvelles métriques ainsi que de nouvelles métaphores de visualisation sont proposées chaque mois. Un outil de visualisation de métriques a donc besoin d'un mécanisme simple pour l'ajout de nouvelles visualisations, d'un mécanisme simple pour définir de nouvelles métriques et surtout, *d'un mécanisme qui permet l'intégration à l'exécution de ces métriques avec ces visualisations*.

Quelques travaux ont été présentés où de puissantes librairies de visualisation de données ont été utilisées pour faire la visualisation des résultats des métriques [27]. Bien que cette façon de faire puisse donner des résultats impressionnants, un certain effort de programmation est nécessaire pour convertir les données dans un format compatible avec la composante de visualisation et ensuite pour générer la visualisation. Les outils présentement disponibles ne fournissent pas de mécanisme qui permettraient à l'utilisateur de créer ses propres visualisations en intégrant lui-même, à l'exécution, des métriques avec des visualisations qui ont été ajoutées dans l'outil. Plusieurs raisons expliquent ce manque de flexibilité et d'extensibilité que l'on retrouve présentement dans la plupart des outils.

Dans un premier temps, comme il existe une grande diversité au niveau des méta-modèles à partir desquels les métriques sont calculées, transporter une métrique d'un certain environnement vers un autre environnement nécessite généralement une ré-implantation logicielle complète et ceci même si un même langage est utilisé. De plus, étant donné l'incompatibilité de ces méta-modèles, en général, la nouvelle implémentation de la métrique ne sera pas fidèle à l'implantation originale et donnera des résultats différents.

Ensuite, il y a la problématique de l'intégration de métriques ayant leur résultats sur un certain type d'échelles avec des variables visuelles de différents niveaux. Par exemple, une métrique qui produit des résultats sur l'échelle des réels ne peut pas être représentée graphiquement par une variable visuelle qui ne serait constituée que de sept différentes couleurs. D'un autre coté, une métrique donnant des résultats sur une échelle nominale sera tout aussi mal représentée par une variable visuelle de taille.

Les différentes variables visuelles sont plus ou moins bien adaptées à la représentation des données sur les différents types d'échelles. Dans certains cas, la représentation est impossible. Un outil de visualisation de métriques vraiment utile devrait donc incorporer une logique qui interdit à l'utilisateur le jumelage de certaines métriques avec certaines variables visuelles et favoriser ce jumelage dans d'autres cas. Dans le même ordre d'idées, on retrouve le problème des bornes. Ce problème vient du fait que plusieurs métriques donnant des résultats sur une échelle continue n'ont pas de bornes à priori alors que toute visualisation a une limite de taille bien précise soit la taille de l'écran de l'ordinateur. Différentes stratégies peuvent être employées pour effectuer la projection des différentes échelles sur les variables visuelles.

Encore plus problématique, il existe un grand nombre de librairies de visualisation très puissantes déjà disponibles sur le marché. Rendre possible leur intégration et leur cohabitation dans un outil capable de calculer et visualiser les métriques est donc une idée très attrayante. Cependant, ces librairies ont toutes des interfaces différentes et acceptent les données à visualiser sous des formats variés. Encore une fois, ceci rend

très difficile l'intégration de ces visualisations et encore plus l'utilisation de celles-ci pour visualiser les résultats des métriques.

Certaines visualisations se complémentent naturellement. Par exemple, il est très utile de visualiser une métrique de taille de classe dans le contexte de l'arbre d'héritage des classes. Cela permet d'identifier rapidement les classes qui sont trop grosses et de visualiser celles-ci dans le contexte de sa position dans l'arbre d'héritage. Mais cette image ne donne qu'un vague aperçu de la distribution réelle des classes par rapport à la taille. Un graphique de densité ou un graphique en tarte complémente bien l'arbre d'héritage. Mais si les deux graphiques viennent de différentes librairies de visualisation, avoir ces visualisation dans la même fenêtre avec les mêmes contrôle et une certaine interaction peut être difficile. Un outil de visualisation de métrique devrait permettre l'intégration de plusieurs visualisations venant de différentes librairies de visualisation et ce, de façon transparente.

4.3 Considérations psychophysiques

Les avancées énormes faites ces dernières décennies dans le monde de la psychophysique trouvent facilement leurs applications dans le domaine de la visualisation d'information et, par conséquent, pour la visualisation des métriques aussi. Plusieurs résultats empiriques de la psychophysique sont directement applicables dans le contexte de notre recherche pour créer des visualisations efficaces.

Cependant, l'ensemble des connaissances psychophysiques qui s'appliquent dans le contexte de la visualisation des métriques est assez vaste et complexe et n'est pas immédiatement accessible à l'utilisateur commun. Au lieu de cela, les principes psychophysiques doivent être incorporés à l'outil qui visualise les métriques. L'utilisateur doit pouvoir bénéficier de ces principes psychophysiques sans avoir à les assimiler.

4.4 Implantation d'une solution réutilisable

Le but premier de ce mémoire est de présenter des solutions réutilisables à certains problèmes reliés à la visualisation des métriques orientés objet. Ces solutions se

veulent applicables dans tous les outils faisant la visualisation de métriques. Pour rendre des solutions de ce genre plus faciles à utiliser, une implémentation logicielle est souvent fournie.

Il existe plusieurs formats sous lesquelles cette implémentation peut être donnée. On peut la donner sous la forme d'un outil fonctionnel qui réalise les solutions proposées. L'outil devient donc une sorte de validation, de preuve de concept, démontrant que les solutions proposées fonctionnent. Cette alternative a comme inconvénient que l'implémentation ne peut pas être directement réutilisée dans d'autres outils. Les utilisateurs désireux de profiter de ces solutions sont forcés d'utiliser l'outil et doivent abandonner leur propre environnement de calcul et de visualisation de métriques.

On peut aussi implémenter les solutions sous la forme d'une librairie, c'est-à-dire un ensemble de fonctionnalités appelées par l'utilisateur à sa guise. Une librairie logicielle est assez facile à implémenter et à tester et est facilement réutilisable. L'inconvénient principal se situe au niveau du contrôle du flot d'exécution. Implémenter une solution sous la forme d'une librairie ne laisse pas beaucoup de contrôle au concepteur de la librairie sur l'ordre dans lequel les différentes fonctionnalités peuvent être appelées. Ceci n'est pas un problème si la librairie propose un ensemble de fonctionnalités assez indépendantes les unes des autres. Mais notre solution est constituée d'un ensemble de fonctionnalités fortement reliées qui doivent être appelées dans un ordre précis et qui dépendent à plusieurs endroits d'autres fonctionnalités fournies par l'utilisateur de la solution.

Étant donné toutes ces contraintes, la meilleure façon d'implémenter la solution proposée dans ce mémoire est à l'aide d'un cadre d'application. Il y a plusieurs avantages à présenter la solution sous cette forme. Premièrement, la solution pourra être réutilisée librement dans d'autres outils. Plus important, puisque le flot d'exécution est contrôlé par le cadre d'application, la bonne utilisation de ce dernier peut être forcée. Notre cadre d'application comprendra plusieurs blocs de code appelant tantôt des fonctionnalités du cadre lui-même et tantôt des fonctionnalités devant être implémentées par l'utilisateur du cadre d'application. Comme expliqué

dans la section 2.3, un cadre d'application est la meilleure façon de s'assurer que ces fonctionnalités seront implémentées par l'utilisateur et qu'elles seront appelées dans le bon ordre.

Les cadres d'applications ont aussi plusieurs inconvénients. Premièrement ils sont difficiles à implémenter et à tester. En général, on ne peut pas être assuré du bon fonctionnement d'un cadre d'application avant qu'il n'ait été utilisé pour implémenter plusieurs applications. Tester le cadre d'application nécessitera l'implémentation d'une application ou d'un semblant d'application. Finalement les cadres d'applications sont difficiles à apprendre pour un utilisateur et nécessitent une meilleure documentation que les librairies.

La solution présentée dans ce mémoire est complexe et demande un énorme niveau de flexibilité pour pouvoir s'adapter à la grande variété de librairies de visualisation disponibles sur le marché. Implémenter cette solution sous la forme d'un cadre d'application nécessite la résolution d'un grand nombre de problèmes de conception.

Chapitre 5 : L'environnement SPOOL

Dans le cadre du projet SPOOL, un environnement portant le même nom a été développé. Beaucoup des fonctionnalités de l'implémentation de la solution proposée dans ce mémoire sont directement héritées de cet environnement. Il sera donc brièvement décrit dans ce chapitre. La première section présente un survol rapide de l'environnement alors que les deux sections suivantes examinent les deux aspects de l'environnement qui sont les plus importants pour nos travaux : le dépôt et les fonctionnalités de visualisation.

5.1 L'environnement SPOOL

L'environnement SPOOL est un environnement de rétro-ingénierie principalement conçu pour faire l'étude des systèmes de grande taille. Il permet d'extraire les données statiques du code source et de les stocker dans un dépôt. Ces données sont ensuite récupérées et manipulées par un ensemble d'outils qui permettent d'étudier le système.

La Figure 9 représente schématiquement le fonctionnement de l'environnement SPOOL.

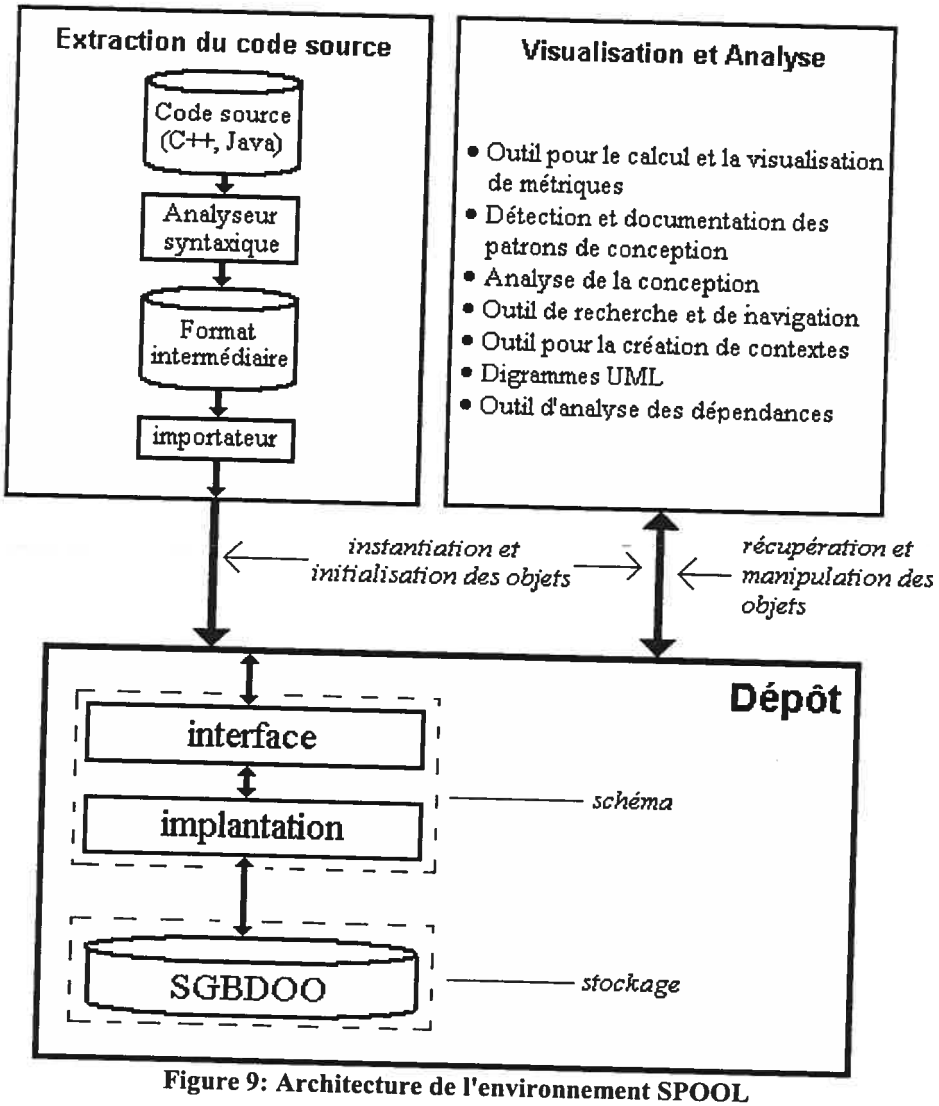


Figure 9: Architecture de l'environnement SPOOL

Le dépôt

Au cœur de l'environnement SPOOL se trouve le dépôt de données. Ce dépôt est constitué sur la base d'un système de gestion de base de données orienté objet (SGBDOO) commercial appelé POET [65]. L'aspect orienté objet de la base de données nous permet de stocker les données extraites sous la forme d'objets Java standards. Le schéma de ce dépôt est lui aussi réalisé en Java et est largement emprunté au méta-modèle UML version 1.1 [83]. Sous sa forme actuelle, le dépôt de SPOOL ne contient pas les détails du code source des systèmes étudiés. Par exemple, l'intérieur du corps des méthodes n'est pas présent, seulement leur signature et différentes relations entre la méthode et d'autres éléments du système sont présentes. Les informations présentes dans le dépôt sont données dans le Tableau 8.

1.	Fichiers [nom, répertoire]
2.	Classes, structures, unions, types primitifs
3.	Hiérarchie d'héritage
4.	Attributs [nom, type, propriétaire, visibilité]
5.	Opérations et méthodes [nom, type, propriétaire, polymorphisme, sorte]
6.	Paramètres [nom, type]
7.	Types de retour [nom, type]
8.	Appels d'opérations [opération, receveur]
9.	Création d'objets
10.	Utilisation de variables
11.	Relations amies (*friendship*)

Tableau 8 : Information contenue dans le dépôt de SPOOL

L'importation des données

Ce dépôt peut être peuplé de différentes façons. La procédure habituelle consiste à utiliser un analyseur syntaxique (*parser*) qui extrait les informations du code source et les place dans des fichiers sous un format intermédiaire. Ensuite, un autre outil, l'importateur, capable de lire ce format intermédiaire, crée les objets et les stocke dans le dépôt. Quatre stratégies différentes ont été utilisées au cours du projet SPOOL.

L'analyseur syntaxique GEN++ [23] a été utilisé pour les système écrits en C++. Il fut utilisé pour générer des fichiers dans un format intermédiaire maison. Un importateur comprenant ce format fut implémenté en Java. Il y a aussi eu des efforts pour prendre avantage des dépôts de données présents dans des outils commerciaux puissants comme l'environnement de développement SNiFF+ [75]. Cet environnement fournit une API permettant d'accéder aux informations se trouvant dans sa table de symboles. Cette table de symboles contient des informations sur le code source.

Finalement, un importateur XMI (XML Metadata Interchange)[90] fut conçu pour SPOOL. La technologie XMI, fut choisie comme format intermédiaire privilégié pour SPOOL à cause de sa popularité et de son adoption par l'OMG comme mécanisme d'échange de données pour les outils de modélisation et les dépôts de données. C'est en passant par cet importateur que les données générées par l'outil Datrix [58], qui sont sous le format d'un ASG (*Abstract Semantic Graph*), ont pu être importés dans

le dépôt de SPOOL. Ceci nécessita la conception d'un convertisseur capable de générer des fichiers au format XMI à partir des fichiers au format d'un ASG générés par DATRIX. D'une façon similaire, le *Information Model* très riche que l'on retrouve dans l'outil Discover[25] fut utilisé comme source d'information pour le dépôt de SPOOL. Dans ce cas, seul un ensemble de requêtes écrites en langage Tcl/Tk fut nécessaire pour générer les fichiers XMI qui purent ensuite être importés dans SPOOL via le même importateur XMI.

Les outils

Plusieurs outils faisant usage des données du dépôt ont été conçus. Comme les données relatives aux systèmes étudiés sont stockées sous forme d'objets Java standards, un outil écrit en Java peut utiliser et manipuler directement ces objets. Les outils peuvent naviguer dans le réseau d'objets représentant le système de façon transparente, le chargement de ces objets depuis la base de données étant pris en charge par le SGBDOO.

L'environnement SPOOL comprend un outil permettant de faire la détection et la documentation des composantes de conception[48], un outil de recherche et de navigation, le *Design Browser* [70][71][27], un outil permettant de faire la mise en contexte d'éléments d'un système, le *Context Viewer* [91], en plus de l'outil permettant de faire le calcul et la visualisation des résultats des métriques orientés objet qui sera présenté au Chapitre 8. Ces outils ont le pouvoir d'ajouter de l'information au dépôt. Par exemple, les composantes de conception identifiés sont stockées dans le dépôt pour éviter que la même requête ne soit à nouveau effectuée ultérieurement. Plus tard, nous verrons que les résultats du calcul des métriques et les perspectives créées par l'utilisateur peuvent eux aussi être stockés dans le dépôt.

5.2 Le schéma du dépôt de SPOOL

Les classes du schéma du dépôt de SPOOL

Le schéma du dépôt de SPOOL a été décrit en détails dans le livre *Advances in Software Engineering* [72] paru récemment. Nous ne donnerons ici qu'une brève description des classes que l'on retrouve dans le dépôt de SPOOL.

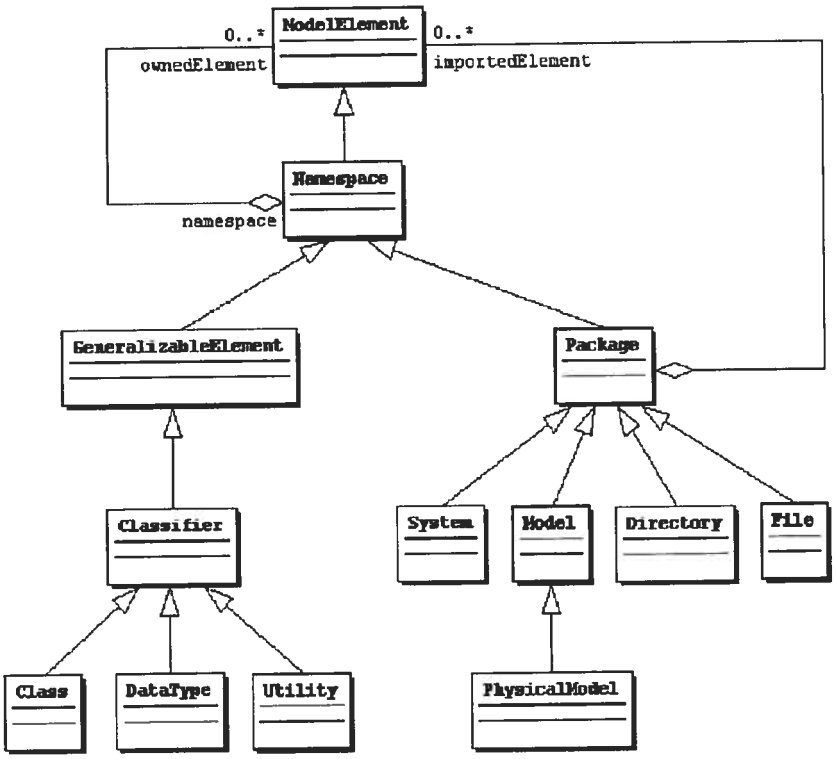


Figure 10: Schéma partiel du dépôt de SPOOL : Les *Namespaces*

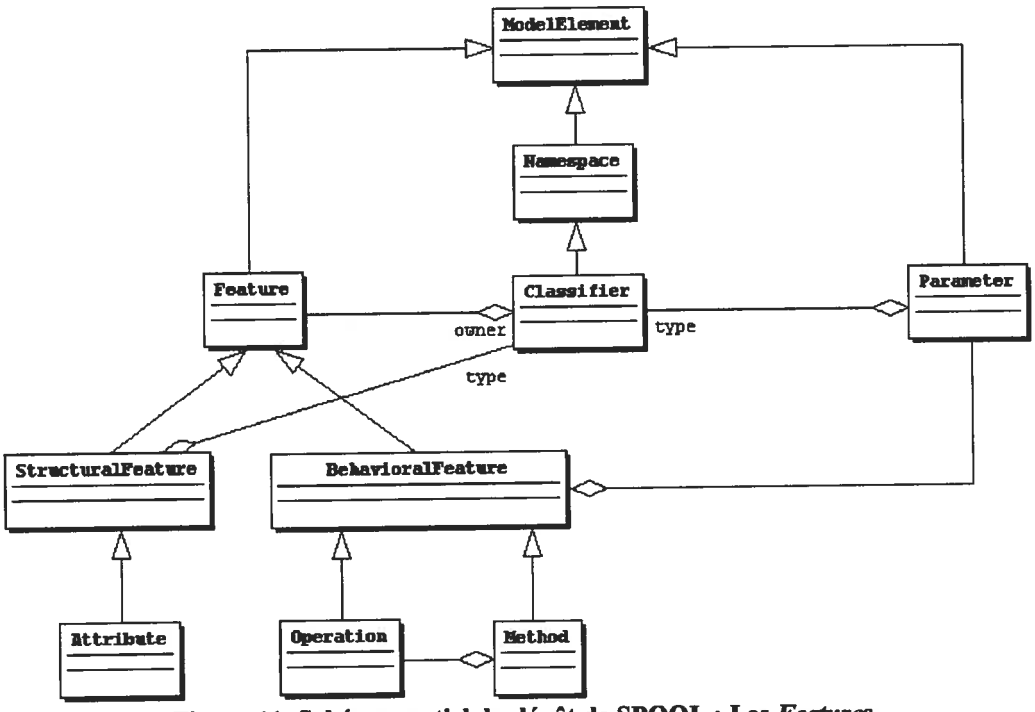


Figure 11: Schéma partiel du dépôt de SPOOL : Les *Features*

La Figure 10 et la Figure 11 décrivent les classes centrales du dépôt de SPOOL. Chacune de ces classes implémente des fonctionnalités, mais seulement les classes

formant les feuilles de l'arbre d'héritage (*Class*, *DataType*, *Utility*, *System*, *PhysicalModel*, *Directory*, *File*, *Attribute*, *Operation*, *Method*, *Parameter*) sont instanciées. Selon les spécifications MOF, les instances de ces classes représentent des éléments d'un modèle de niveau M1, c'est pourquoi les instances des classes du méta-modèle sont appelées des *ModelElement*s. Nous donnons ici une brève description du rôle des différentes classes de la Figure 10 et de la Figure 11 :

- Un *ModelElement* est la classe racine du schéma du dépôt de SPOOL. Un *ModelElement* est une abstraction d'un élément quelconque du système étudié.

- Un *Namespace* contient un ensemble de *ModelElements* dont le nom est unique à l'intérieur du *Namespace*.

- Un *GeneralizableElement* est un *ModelElement* qui peut participer à une relation de généralisation comme une relation d'héritage de classe par exemple.

- Un *Classifier* est un *ModelElement* qui définit un ensemble de propriétés structurelles et comportementales. Une classe, par exemple, est un *Classifier* ; elle définit ses propriétés structurelles (ses attributs), et ses propriétés comportementales, (ses méthodes). À part la classe *Class*, la classe *Classifier* a deux autres sous-classes concrètes : *DataType* qui représente les types de données primitives (int, float, char, etc..) et *Utility* qui est un concept qui permet de classifier les exceptions au paradigme orienté objet comme les variables globales et les fonctions libres.

- Un *Feature* est une propriété d'un *Classifier*, elle peut être comportementale (*BehavioralFeature*) ou structurelle (*StructuralFeature*). La seule sous-classe de *StructuralFeature*, *Attribute* représente un attribut d'une classe au sens commun de la programmation orientée objet. Ici, il faut bien faire la distinction entre les deux types de *BehavioralFeature,* i.e., *Method* et *Operation*. *Operation* ne représente que la signature d'une méthode (*Method*)

et contient une liste de paramètres (*Parameter*). Cette opération peut être implémentée ou non par une méthode (*Method*) dans la classe où elle est définie et dans les sous-classes de cette classe. À cause du polymorphisme, à l'exécution, un même appel d'opération peut résulter en un appel à différentes méthodes dépendamment du type de l'objet sur lequel l'opération est appelée.

- Un *Package* est un ensemble de *ModelElement*. *System* est le plus vaste des packages, il représente le système étudié lui-même. Un *System* contient un *PhysicalModel* qui est une représentation du modèle physique du système, c'est-à-dire les répertoires (*Directory) et* les fichiers (*Files*) qui eux contiennent les éléments du code source proprement dit, les classes (*Class*), les *Utility* et autres.

Comme mentionné précédemment, le dépôt de SPOOL ne contient pas tous les détails du corps des méthodes. Au lieu de cela, chaque méthode contient un ensemble d'*Action*s. UML définit une action comme étant un traitement atomique dont le résultat est de changer l'état du système et/ou de retourner une valeur. Dans le dépôt de SPOOL, les *Action*s sont utilisées pour représenter différentes instructions pouvant se trouver à l'intérieur du corps d'une méthode.

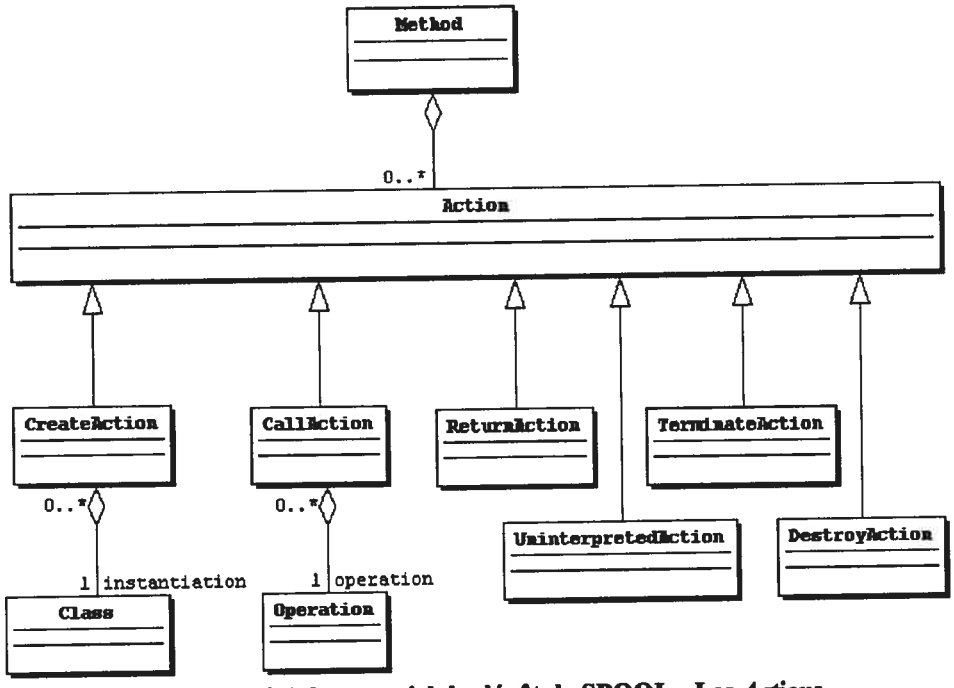


Figure 12: Schéma partiel du dépôt de SPOOL : Les *Actions*

La Figure 12 illustre quelques-unes de ces actions. Par exemple une des sous-classes de la classe *Action*, *CreateAction*, permet de représenter l'instantiation d'un objet. Cette classe contient entre autres une référence vers le type de l'objet instancié. Les appels de méthodes sont aussi représentés via la classe *CallAction* qui contient une référence vers l'opération (*Operation*) correspondant à l'appel de la méthode (*Method*). Les autres types d'action (*ReturnAction*, *TerminateAction*, *UninterpretedAction*, *DestroyAction*) ne sont présentement pas utilisés dans SPOOL en raison d'un compromis fait au niveau de la performance.

5.3 La visualisation dans l'environnement SPOOL

Mise a part les fonctionnalités présentes dans la schéma de son dépôt, SPOOL fournit un vaste ensemble de fonctionnalités visant à faciliter la création d'outils CASE. Bien que cela ne soit pas une obligation, tous les outils créés dans le cadre du projet SPOOL utilisent ces fonctionnalités. En fait, l'environnement SPOOL peut être vu comme étant un cadre d'application pour la création d'outils CASE. Ces fonctionnalités sont diverses et ont été partiellement décrites dans une thèse de maîtrise [57]. Une grande portion de ces fonctionnalités sont vouées à la visualisation et sont utilisées pour l'implémentation de la solution présentée dans ce mémoire.

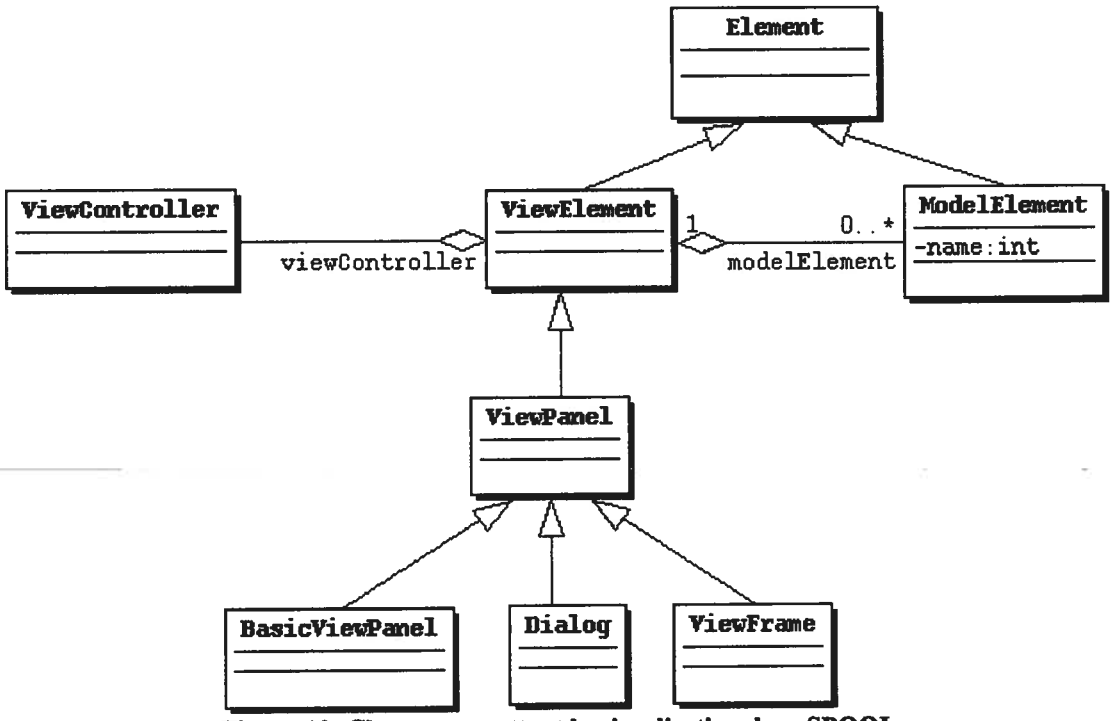


Figure 13: Classes permettant la visualisation dans SPOOL

La Figure 13 montre quelques-unes des classes qui implantent ces fonctionnalités. On note la classe *Element* qui est la classe racine de toute la hiérarchie de classes de SPOOL, tant du coté du méta-modèle que du coté des outils. Cette classe implémente la patron de conception *Observer* (voir [37] p. 293), qui permet en principe à tous les objets dans SPOOL d'*observer* n'importe quel autre objet, c'est-à-dire d'être avertis lorsque des changements se produisent dans celui-ci.

Un *ViewElement* est une représentation graphique d'un ensemble de *ModelElement*. Cette classe est la super-classe de toutes les classes des outils de SPOOL. Elle fournit les fonctionnalités relatives aux interfaces usagers. Un *ViewElement* tient une référence à l'ensemble de *ModelElement* qu'il représente, conformément au patron architectural *MVC* [15]. L'aspect *Controller* du patron *MVC* est réalisé par la classe *ViewController*. Grâce au mécanisme d'observation, lorsque les données contenues dans un *ModelElement* changent, les *ViewElements* qui y sont associés modifient leur apparence conformément à ces changements. L'environnement SPOOL fournit donc une synchronisation entre les interfaces usagers des outils et les données se trouvant dans le dépôt.

Un *ViewPanel* est une surface quelconque de l'interface usager. Il peut contenir d'autres *ViewPanels* ou des éléments graphiques atomiques comme des *widgets* ou des éléments du diagramme de classes (voir patron *Composite* dans [37] p.163). Pour implémenter une nouvelle visualisation profitant de tous les mécanismes de SPOOL décrits ci-haut, il suffit donc d'implémenter une sous-classe de *ViewPanel*.

La classe *ViewFrame* implémente les fonctionnalités d'une fenêtre d'une interface usager standard avec une barre de menus et une barre d'outils. L'environnement SPOOL s'assure que de la synchronisation modèle/vue se répercute à travers tous ces *ViewFrame*s.

Chapitre 6 : Concepts sous-jacents à MDP

Ce chapitre vise à décrire de façon précise les concepts centraux du cadre d'application MDP. La section 6.1 définit précisément le concept de métrique dans MDP. Cette section établit également un nouveau vocabulaire qui sera utilisé dans le reste de cet ouvrage pour décrire les métriques. La section 6.2 explique ce qu'est un diagramme dans MDP alors que la section 6.3 fait de même pour le concept de perspective.

6.1 Métriques

6.1.1 Langage de définition des métriques

Comme expliqué au Chapitre 4, la condition première pour permettre que les résultats des métriques soient réutilisables est de donner aux métriques une définition précise et formelle. Il faudra donc définir ou choisir un langage approprié à cette tâche.

Sémantique du langage

Le plus gros problème à la définition d'un tel langage se situe au niveau de sa sémantique. Le vocabulaire utilisé pour décrire les métriques doit être précis et ne laisser aucune place à l'interprétation. La façon la plus sûre d'éviter tout écart sémantique entre les termes utilisés dans les définitions des métriques et les méta-éléments du méta-modèle utilisé est de *définir les métriques directement en fonction de ces méta-éléments*.

Théoriquement, cette façon de faire résout le problème décrit par la Figure 8, celui de la définition des métriques. L'ensemble des méta-éléments du méta-modèle UML par

exemple, peut être utilisé pour définir des métriques de design. Cependant, pour plusieurs raisons, le méta-modèle UML est un choix très discutable. Premièrement, comme mentionné à la section 4.1, la documentation de UML ne spécifie pas de projection de la sémantique de UML vers les différents langages de programmation. La façon précise par laquelle les différents concepts des langages de programmation sont adaptés à UML et/ou vice-versa est laissée à la discrétion de l'utilisateur et laisse énormément de place à l'interprétation. Ensuite, UML ne possède pas toute la sémantique nécessaire pour modéliser le détail de l'intérieur du corps des méthodes et bien qu'il y ait plusieurs façons de contourner ce problème (voir [77] section 3.1), on a encore une situation qui laisse place à l'interprétation. Finalement, même si on s'en tient à la sémantique du package *Behavioral Elements*, la façon complexe par laquelle UML modélise des choses simples comme les appels de méthodes rend le méta-modèle UML peu approprié pour des tâches algorithmiques complexes comme le calcul de métriques[72][23]. Pour toutes ces raisons, bien que UML fasse presque l'unanimité au niveau des outils de développement (voir [77] section 3.2) pour les outils orientés vers les activités de maintenance, UML est assez peu utilisé (voir [77] section 3.3).

Il nous apparaît cependant que la sémantique utilisée dans le langage de définition des métriques doit correspondre à celle du méta-modèle utilisé pour modéliser le système et non à celle du langage de programmation dans laquelle le système est conçu.

Syntaxe du langage

Trouver une syntaxe pour ce langage est un moindre problème. Par exemple, un langage naturel pourrait être utilisé en conjonction avec la sémantique fournie par le méta-modèle pour fournir une description non-ambiguë de la métrique. Cependant, cette façon de procéder rend impossible l'utilisation directe de la définition de la métrique par un outil en plus de laisser ouverte la possibilité d'une mauvaise interprétation par le programmeur qui implémenterait la métrique. *Pour ces deux raisons, le langage de définition des métriques se doit d'être formel.*

6.1.2 Type d'ensemble de départ des métriques

Les métriques sont des mesures faites sur une représentation du système, i.e., sur un modèle du système. Concrètement, une métrique sera calculée à partir d'un petit nombre de *ModelElements* ayant chacun un type précis. Par exemple, chaque mesure faite avec la métrique WMC, *Weighted Method for Class*, est associée à un et un seul *ModelElement* de type *Class*. Dans certains cas, la mesure dépend de deux ou trois *ModelElement*s ou même plus. On peut par exemple imaginer une métrique CBC, *Coupling between Class*es, mesurant le couplage entre deux classes. Une telle métrique dépend donc de deux *ModelElements* de type *Class* à la fois. Il existe aussi des métriques pertinentes qui dépendent de trois *ModelElements* ou plus, mais ce genre de métriques est plus rare dans la littérature.

Rappelons qu'au Chapitre 2 nous avons défini une métrique comme étant une mesure au sens de la théorie de la mesure c'est-à-dire, un homomorphisme entre un ensemble d'objets empiriques et un ensemble d'objets mathématiques formels. Le paragraphe précédent démontre que pour une métrique donnée, *l'ensemble d'objets empiriques sera constitué des tuples formés par le produit cartésien multiple de un ou plusieurs ensembles de ModelElements, chaque ensemble contenant des ModelElements de même type, i.e., qui sont des instances d'un même méta-élément. Le langage de définition des métriques doit incorporer cette réalité.*

Définition 10 : Type d'un tuple

Dorénavant, la notation suivante sera utilisée pour exprimer le type d'un de ces tuples :

[méta-élément-1,méta-élément-2,...,méta-élément-n] ,

où $n \geq 1$. Les types de tuple suivants sont représentatifs :

[Class] [Method] [Class,Class] [Class,Package] [Method,Class] [Method,Generalization] [Package,Class,Feature]

(les méta-éléments mentionnés sont présents dans le méta-modèle UML).

Définition 11 : Degré d'une métrique

Par simplicité, on définit le degré d'une métrique comme étant le nombre de *ModelElement*s à partir de laquelle elle est calculée, i.e., la cardinalité du tuple. Donc, les métriques calculées à partir d'un seul *ModelElement* sont dites des métriques *de degré 1* ou *.du premier degré* (ex : **[Class]**), celle qui sont calculées à partir d'un couple de *ModelElement*s sont dites *de degré 2* ou *du deuxième degré* (ex : **[Method,Class]**) et ainsi de suite.

La plupart des métriques que l'on trouve dans la littérature sont des métriques du premier degré. On a donc ici, le cas trivial du produit cartésien d'un seul ensemble, le résultat étant l'ensemble lui-même. On considérera chaque élément de cet ensemble comme des tuples à un élément. Pour les métriques du deuxième degré, le produit cartésien produira un ensemble de tuples à deux éléments, pour les métriques du troisième degré, des tuples à trois éléments et ainsi de suite.

Restrictions sur les ensembles de *ModelElement*s

Pour les métriques du premier degré, le plus souvent, l'ensemble d'objets empiriques s'étendra à tous les *ModelElement*s du système d'un certain type. La métrique WMC par exemple peut être calculée sur n'importe quelle classe d'un système. Cependant, la définition d'une métrique peut poser certaines restrictions quant aux *ModelElement*s qui se trouvent dans les ensembles, particulièrement dans le cas des métriques de degré 2 ou plus. Ainsi, on peut imaginer une métrique NAA, *Number of access to attribute*, qui calculerait le nombre d'accès qu'une méthode fait à un attribut défini à l'intérieur de la même classe. Cette métrique du deuxième degré dépend d'un *ModelElement* de type *Method* et d'un autre de type *Attribute* (en assument que ces deux méta-éléments existent dans le méta-modèle utilisé). Elle sera donc calculée à partir d'un tuple de type **[Method, Attribute]**. Cependant, le produit cartésien ne sera pas celui de l'ensemble de toutes les méthodes du système par l'ensemble de tous les attributs du système car on ne s'intéresse qu'aux couples méthode/attribut définis dans la même classe. Au lieu de cela, le produit cartésien sera filtré par la condition que la méthode et l'attribut de chaque couple doivent être définis dans la même

classe, une condition qui restreint énormément le nombre d'éléments dans le résultat du produit cartésien. Ces contraintes peuvent être de tout genre. *Le langage de définition des métriques doit également permettre de spécifier ces contraintes.*

6.1.3 Type d'ensemble d'arrivée des métriques

Comme mentionné dans le Chapitre 2 une métrique est une mesure qui peut rendre ses résultats sur différents types d'échelles et donc, différents ensembles mathématiques sont nécessaires pour exprimer les résultats. Pour les échelles de type absolu, le seul choix logique est l'ensemble des nombres naturels ($\mathbb{N}$) car seul le dénombrement d'un ensemble d'objet constitue une mesure sur une échelle de type absolu. Dans le cas des échelles de type ratio et intervalle, l'ensemble des nombres rationnels s'impose ($\mathbb{Q}$) car c'est un ensemble mathématique connu et parce qu'il possède les propriétés désirées (voir Tableau 1). Pour les échelles de type nominal et ordinal, une mesure consiste à classer les objets de la mesure dans des catégories. Dans le premier cas, les catégories ne sont pas ordonnées, dans le deuxième cas elles le sont. La meilleure façon d'étiqueter ces catégories c'est à l'aide de chaînes de caractères. Pour les échelles de type nominal, on voudra que les chaînes de caractères expriment la nature de chaque catégorie. Pour les échelles de type ordinal, il est préférable que les chaînes de caractères aient en plus des significations qui permettent de classer les catégories en ordre. Nous attribuerons les symboles

ID {chaîne-1, chaine-2, ... , chaine-n }

et

ID-O { chaîne-1, chaine-2, ... , chaine-n }

respectivement aux ensembles de ce type. Le Tableau 9 donne quelques exemples pour chacun des types d'échelles.

Type d'échelles	Ensemble mathématique	Symbole	Exemples
Nominal	Ensemble de chaînes de caractères	***ID{...}***	{helper_class, abstraction_class, concrete_class}, {AbstractClass, ConcreteClass, aucun}
Ordinal	Ensemble de chaînes de caractères ordonnées	***ID-O{...}***	{mauvais, bon, très bon, excellent}, {petit, moyen, large, très large}
Intervalle	Rationnels	***Q***	-18.23, 90.0, 345.786
Ratio	Rationnels	***Q***	-18.23, 90.0, 345.786
Absolue	Naturels	***N***	2, 3, 15, 23 456

Tableau 9: Ensembles mathématique utilisés pour chaque type d'échelles

Pour le type d'échelles nominal, on remarque que les deux ensembles données en exemple n'ont pas d'ordre naturel. Le premier exemple pourrait servir d'échelle pour une métrique ROC, *Role Of Class*, qui tente de déterminer le rôle prédominant d'une classe, soit une classe contenant des fonctions utilitaires (helper_class), une classe utilisée principalement comme abstraction (abstraction_class) ou, une classe concrète à partir de laquelle des objets sont instanciées (concrete_class). Le deuxième exemple pourrait servir d'échelle à une métrique qui vérifie si la classe semble jouer l'un des rôles du patron de conception *Template Method* ([37] p.325). Pour le type d'échelles ordinal, on peut facilement voir que les éléments de l'ensemble ont des significations qui permettent de les mettre en ordre.

Le langage de définition des métriques doit supporter les quatre types de résultats du Tableau 9, soit les ensembles de chaînes de caractères (***ID{...}***),*les ensembles de chaînes de caractères ordonnées* (***ID-O{...}***), *les nombres rationnels* (***Q***) *et les nombres naturels* (***N***).

6.1.4 Définition d'une métrique

Type d'un métrique

Le type d'une métrique dépend à la fois du type de son ensemble de départ et du type de son ensemble d'arrivée. On parle donc ici du type du tuple et du type d'un ensemble d'objets mathématiques.

Définition 12 : Type d'une métrique

Le type d'une métrique est un couple (A,B) où A est le type des tuples de l'ensemble de départ et B est l'ensemble d'arrivée tel décrit dans le Tableau 9.

Par exemple, un type approprié pour la métrique WMC serait (**[Class]**,***Q***) alors qu'un bon type pour la métrique NAA serait (**[Method,Attribute]**,***N***). Finalement, un type approprié pour la métrique ROC serait

([Class], *ID{ helper_class, abstraction_class, concrete_class }*)

Définition précise d'une métrique

Il est maintenant temps de regrouper toutes les considérations faites jusqu'à présent au sujet de la façon de définir des métriques et qui doivent être prises en considération par le langage de définition. La Définition 13 ci-dessous est une définition du concept de métrique dans le cadre de son environnement technique. Cette définition prend en compte l'exigence qu'une métrique doit être une mesure au sens de la théorie de la mesure (voir Définition 9), elle se veut les grandes lignes d'une solution aux problèmes exprimés par les figures de la sections 4.1 (Figure 6, Figure 7 et Figure 8) et elle incorpore plusieurs des considérations faites depuis le début de ce chapitre quant au langage de définition. Bien que cette définition ne soit pas vraiment formelle, nous verrons après qu'en encadrant celle-ci dans des technologies concrètes, on peut obtenir une définition complètement formelle ou presque.

Définition 13 : Métrique (dans un environnement technique quelconque)

Soit

MM, un méta-modèle

$\mathbf{E_{MM}}$, un méta-élément faisant partie du méta-modèle **MM**

L, un langage de programmation

$\mathbf{P_{L->MM}}$, une projection de la sémantique du langage **L** dans le méta-modèle **MM**,

$\mathbf{S_L}$, un système logiciel conçu dans le langage de programmation **L**

$\mathbf{M_S}$, un modèle du système S_L, (donc une instance du méta-modèle **MM)**

$\mathbf{E_{Ms}}$, un *ModelElement* qui est une instance du méta-élément $\mathbf{E_{MM}}$ et qui fait partie du modèle $\mathbf{M_s}$

De plus, soit l'ensemble

$$W = \prod_{i=1}^{n} G_i$$

où Π est le produit cartésien multiple et où les G_i sont des ensembles définis de la façon suivante :

$$G_i = \{ E_{M_s} \mid E_{M_s} \text{ est une instance de } E_{MM\,i} \}$$

Le contenu des ensembles G_i peut cependant être restreint par un ensemble de contraintes supplémentaires. Ces contraintes posent des limitations sur l'applicabilité de la métrique. Maintenant soit le système relationnel empirique $\Im$ (voir Définition 2)

$$\Im = (W, R_1, R_2, \ldots , R_p, \mathbf{o_1}, \mathbf{o_2}, \ldots , \mathbf{o_q}),$$

et le système relationnel formel $\wp$ (voir Définition 3)

$$\wp = (F, S_1, S_2, \ldots , S_p, \bullet_1, \bullet_2, \ldots , \bullet_q)$$

avec l'homomorphisme μ, de $\Im$ à $\wp$. Si $(\Im, \wp, \mu)$ est une échelle d'un des types présents dans le Tableau 1, alors μ est une métrique de type

$$([E_{MM1}, E_{MM2}, \ldots, E_{MMn}], F)$$

de degré n qui est applicable sur un modèle M_S d'un système S_L créé grâce à la projection $P_{L\text{-}MM}$.

Conséquences

Les conséquences de la Définition 13 sont multiples.

- Une métrique est définie en fonction d'un méta-modèle particulier.

- Une métrique est définie en fonction d'un langage de programmation particulier.

- Une métrique est définie en fonction d'une projection particulière entre la sémantique du langage de programmation et le méta-modèle.

- L'application d'une métrique se fait sur un tuple de *ModelElement*s. Les *ModelElement*s à la même position dans chaque tuple sont de même type.

- Les ensembles G_i sont, par défaut, l'ensemble de tous les *ModelElement*s d'un certain type dans le système, mais ces ensembles peuvent être restreints par un ensemble de contraintes nécessaires à la bonne définition de la métrique.

- Bien que théoriquement, le système relationnel formel d'une mesure peut être composé de n'importe quel ensemble d'objets mathématiques imaginables (Vecteurs, Matrices, etc), pour des raisons de simplicité, ici on restreint la définition de métrique au mesure qui peuvent s'exprimer sur l'un des cinq échelles du Tableau 1.

Aspects non-formels de la Définition 13

Plusieurs aspects de la Définition 13 sont volontairement laissés informels car ces aspects sont mieux définis à l'aide d'une technologie concrète. Entre autres, on peut se demander :

- Qu'est qu'un méta-modèle ?

- Qu'est qu'un méta-élément ?

- Qu'est qu'une *projection* de la sémantique d'un langage de programmation dans un méta-modèle. Comment cette projection est-elle spécifiée ?

- Quelles sont les contraintes sur les ensembles de *ModelElements* ? Qu'est-ce qui est permis ? De quelle façon sont-elle spécifiées ?

- Avec quel langage l'homomorphisme, i.e., la métrique elle-même, est-elle spécifiée ?

MOF et OCL, une solution presque complète

L'architecture de méta-modélisation MOF et le langage de contraintes OCL présentés à la section 2.4 comblent la plupart des lacunes de la Définition 13 présentée dans la section précédente. En plus, ces deux technologies fournissent un grand nombre de solutions très pratiques pour l'implémentation logicielle d'un mécanisme pour calculer des métriques qui répondent à la Définition 13.

Dans un premier temps, MOF, à l'aide de son méta-métamodèle donne une définition précise de ce qu'est un méta-modèle et de ce que sont les méta-éléments. Puisque le MOF est capable de définir n'importe quel méta-modèle, tous les méta-modèles existants pourraient être redéfinis en fonction de MOF. On pourrait alors, dans la Définition 13 définir un méta-modèle comme étant une instance de MOF, rendant formel cet aspect de la définition.

Le méta-élément *StructureType* du méta-métamodèle MOF est défini comme suit dans les spécifications ([62] p.3-37)

> "The StructureType class is a type constructor for MOF structure data types. A structuretype is a tuple type (i.e., a cartesian product) consisting of one or more fields. The fiels are defined by StructureField instances contained by the StructureType instance."

En comparant cette définition avec la Définition 13, on voit que le méta-élément *StructureType* peut être utilisé pour créer le type de tuple nécessaire à chaque métrique. L'architecture MOF fournit aussi un package *PrimitiveTypes* qui définit un ensemble de types primitifs de base. Tout méta-modèle basé sur MOF doit supporter ces types de base. Deux de ces types primitifs, *Integer* et *Float* peuvent être utilisés pour modéliser, respectivement, les ensembles mathématiques des naturels ($\mathbb{N}$) et des rationnels($\mathbb{Q}$) qui nous sont nécessaires. Pour modéliser les ensembles de chaînes de caractères ordonnées et non-ordonnées (***ID{...}*** et ***ID-O{...}***) il faut s'en remettre

au type primitif *String* en conjonction avec le méta-élément *CollectionType* défini comme suit par MOF:

> "The CollectionType class is a type constructor for MOF collection types. A collection type is a data type whose values are finite collections of instances of some base type. The base type for a collection data type is given by the CollectionType instance's 'type' value. The 'multiplicity' Attribute gives the collection type's lower and upper bounds, and its orderedness and uniqueness properties."

Il s'agit donc de créer deux nouveaux types de collections ayant pour type de base le type primitif *String* du package *PrimitiveValue*. Le premier type de collection sera non-ordonné et servira à modéliser les résultats qui sont sur une échelle de type nominal alors que le deuxième sera ordonné et servira à modéliser les résultats qui sont sur une échelle de type ordinal.

Donc, MOF nous fournit une façon de modéliser les tuples de départ ainsi que les résultats des métriques de façon cohérente pour tous les méta-modèles. C'est-à-dire que pour tout méta-modèle défini avec MOF, il suffit de créer certains méta-éléments additionnels, soit une instance de *StructureType* par métrique et deux instances de *CollectionsType*, et les résultats des métriques pourront être inclus dans les modèles des systèmes.

Malgré tous ses avantages, MOF ne fournit pas de méthode ni de langage pour définir précisément et formellement des métriques. C'est ici qu'intervient le langage OCL. Utiliser le langage OCL pour définir les métriques semble être l'approche la plus prometteuse présentement. Puisque OCL s'adapte parfaitement à tous les méta-modèles définis à l'aide de MOF, il est un langage parfait pour spécifier les contraintes sur les tuples de l'ensemble de départ. Un peu moins évident est la compétence de OCL pour définir les métriques elles-mêmes. Mais comme le démontre Abreu dans des travaux récents [3], une expression OCL peut être utilisée

pour donner une définition formelle et précise aux métriques et peut en même temps être utilisée pour spécifier les contraintes sur l'applicabilité de la métrique.

Plus précisément, les *pré-conditions* de OCL sont utilisées pour spécifier les contraintes sur l'applicabilité de la métrique en posant des conditions sur les tuples de l'ensemble de départ. Les pré-conditions sont obligatoirement des expressions OCL de type booléen qui doivent être satisfaites. La métrique elle-même est spécifiée à l'aide d'une *post-condition*. Les post-conditions sont des expressions OCL qui, elles, peuvent être de n'importe quel type parmi le vaste ensemble de types définis dans les spécifications OCL. Entre autres les types de bases *Integer*, *Real*, *String*, *Set*, *Sequence* conviennent pour modéliser les ensembles mathématiques du Tableau 9 et peuvent donc exprimer les résultats des métriques.

Choix du méta-modèle et spécification de la projection

Comme expliqué plus haut, les différents langages de programmation sont souvent adaptés librement aux méta-modèles. C'est le cas de UML où cette adaptation est complètement laissée à la discrétion des utilisateurs, des concepteurs d'outils de développement et des concepteurs d'outils de maintenance. Ces projections sont souvent mal documentées et différentes les unes des autres. La Définition 13 implique que les résultats des métriques ne sont pas partageables entre les environnements techniques utilisant différentes projections. Donc, dans un monde dominé par UML dans son état actuel, une métrique définie selon la Définition 13 serait peu réutilisable.

Deux solutions existent à ce problème. Premièrement, on peut tenter de définir un standard pour les projections et tenter de le faire accepter par les gens du domaine. On pourrait par exemple prendre le méta-modèle UML, développer une projection standard et formelle pour les langages de programmation orientés objet les plus populaires (Java, C++, C#, Smalltalk, ...) et proposer celles-ci comme des standards.

Mais puisque la Définition 13 implique qu'une métrique ne peut être définie que pour un seul langage de programmation à la fois, il reste peu de raisons pour vouloir utiliser un méta-modèle indépendant de tout langage pour calculer les métriques. Au

lieu de cela, l'idéal pour faire le calcul des métriques serait de rendre nul l'écart sémantique entre les méta-éléments du méta-modèle et les éléments du langage de programmation. On parle donc de définir un méta-modèle directement à partir des spécifications du langage de programmation. On obtiendrait donc un méta-modèle Java, un méta-modèle C++ et ainsi de suite. Il y a deux avantages principaux à une telle approche. Premièrement, la définition des métriques serait beaucoup plus simple puisque les noms des méta-éléments seraient les mêmes que ceux des concepts sémantiques du langage de programmation qu'ils représentent. Deuxièmement, construire une projection entre le langage et le méta-modèle serait triviale. St-Denis (voir [77] section 7.2.1), après avoir adapté UML à différents langages de programmation orientés objet et fonctionnels, semble en venir à la même conclusion. Et c'est dans cet esprit que NetBeans[61], une compagnie se spécialisant dans la création d'outils de développement logiciels d'avant-garde, s'est lancée dans la définition d'un méta-modèle Java basé sur l'architecture MOF.

MDP avec MOF et OCL

MOF et OCL ont été choisis comme technologie derrière la définition des métriques dans MDP. De toutes les raisons mentionnées ci-haut, le fait que ces deux technologies puissent être adaptées à n'importe quel méta-modèle est sans doute la plus importante. La parenté de ces deux technologies avec le standard UML est aussi un avantage tangible.

Définition 14 : Métrique dans MDP

Une métrique dans MDP est une métrique au sens de la Définition 13 et dont

1) le méta-modèle est une instance de MOF,

2) les contraintes sur le tuple de départ sont spécifiées à l'aide de pré-conditions en langage OCL,

3) la fonction de l'homomorphisme est spécifiée à l'aide d'une post-condition en langage OCL, et dont

4) la projection $\mathbf{P}_{L->MM}$ est spécifiée d'une façon formelle.

Encore une fois cette définition n'est pas complètement formelle à cause de la dernière proposition à propos des projections. Il n'existe pas à notre connaissance de méthode simple pour spécifier ce type de projection.

6.1.5 Exemple complet de la définition d'une métrique avec MDP

On donne ici un exemple complet de la définition d'une métrique dans MDP. Pour définir la métrique elle-même, le langage OCL est utilisé mais, à un niveau de MOF différent des exemples de la section 2.4.3. Alors que les exemples de la section 2.4.3 constituent des contraintes sur les instances des éléments du niveau M1, les métriques de MDP s'appliquent sur les instances des éléments du niveau M2, c'est-à-dire sur les classes, les méthodes et les attributs et autres éléments d'un système. Conformément à la Définition 13, la définition de la métrique inclut une référence au méta-modèle auquel elle est associée, elle spécifie le langage de programmation pour laquelle la métrique est conçue, et elle contient une référence à la projection utilisée pour adapter le langage au méta-modèle.

Cette définition a pour but d'être utilisée directement par des outils qui calculent des métriques. C'est pourquoi des balises XML sont utilisées pour faciliter l'implantation d'outils capables de faire l'analyse syntaxique des définitions des métriques. La métrique définit se nomme NOAIC soit , *Number of attributes per Class*.

=== Début de la définition ===

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <metric-definition>
<formal-name>ca.umontreal.iro.NOAIC </formal-name>
<name>Number of attributes in class</name>
<short_name>NOAIC</short_name>
<url>http://www.iro.umontreal.ca/labs/gelo/spool/metrics/NOAIC.html</url>
<meta-model>UML_1.4</meta-model>
<url-mm>http://www.omg.org/uml/</url-mm>
<language>java</language>
```

<mapping>Standard_java_language_mapping</mapping>
<description>Computes the number of non-static attributes, with any visibility kind.</description>
<tupleType>[Class]</tuple Type>
<scaleType>Absolute</scaleType>
<pre-condition></pre-condition>
<metric>
context ClassTuple ::NOAIC( ) : Integer
post : self.class1.feature>
select(f : Feature | f.oclIsOfKind(Attribute))- >size( )
</metric>
</metric-definition>

=== Fin de la définition ===

La définition de la métrique est stockée seule ou avec d'autres définitions dans un fichier texte appelé fichier de définitions de métriques. Les balises suivantes sont présentes dans la définition de la métrique NOAIC:

<formal-name>. Le nom formel de la métrique est contenu dans cette balise. Pour s'assurer que chaque nom est unique à travers toutes la communauté des métriques, le nom de la métrique est toujours complémenté du nom de domaine de l'institut qui a créé la métrique. Cette méthode est entre autres utilisée dans le monde de la technologie Java pour s'assurer que chaque package a un nom unique.

<name>. Le nom commun de la métrique. Typiquement composé de plusieurs mots, le contenu de cette balise donne une idée générale de ce que calcule la métrique.

<short-name>. Un nom court et pratique pour la métrique. Souvent, ce sera l'acronyme du contenu de la balise <name>. (Ex : WMC pour Weighted Method for Class, CBO pour Coupling Between Object)

<url>. Un lien internet où se trouve plus d'information sur la métrique.

<méta-modèle>. Le méta-modèle pour lequel la métrique est définie.

<url-mm>. Un lien internet ou se trouve plus d'information sur ce méta-modèle.

<langage>. Le langage de programmation pour lequel la métrique est définie.

<mapping>. Un identificateur qui réfère à une définition formelle de la projection utilisé pour adapter le langage de programmation au méta-modèle.

<description>. Une description en langage naturel de ce que la métrique calcule.

<tupleType>. Une description en langage naturel de ce que la métrique calcule.

<scaleType>. Le type de l'échelle de la métrique: *Nominal*, *Ordinal*, *Interval*, *Ratio* ou *Absolute*. Pour les types d'échelle Nominal et Ordinal, un ensemble de chaîne de caractères doit aussi être décrit pour spécifier l'échelle. Par exemple,

Nominal : { helper_class, abstraction_class, concrete_class } ou
Ordinal :{ mauvais, bon, très bon, excellent }

sont des échelles correctement spécifiées.

<pre-condition>. Une pré-condition OCL qui définie les contraintes d'applicabilités sur le tuple.

<metric>. L'expression OCL qui permet de calculer la métrique elle-même.

Dans l'expression OCL, *ClassTuple* est un méta-élément instance de l'élément *StructureType* du méta-métamodèle MOF (voir section 6.1.4). Il correspond au type de tuple de la métrique. Il doit être ajouté au méta-modèle s'il ne l'est pas déjà. Les instances de la classe *ClassTuple* sont des tuples avec un seul *ModelElement* de type *Class*. La méthode que nous utiliserons pour nommer les classes qui implémentent les types de tuples consiste à énumérer les méta-éléments en les séparant d'un 'x' majuscule avec, à la fin, le mot 'Tuple'. Pour le type de tuple **[Class]** on a donc *ClassTuple*, pour le type de tuple **[Class,Class]** on a *ClassXClassTuple* et pour le type de tuple **[Method,Attribute]** on a *MethodXAttributeTuple*. La classe *ClassTuple* a un

attribut *class1* de type *Class* qui permet de récupérer le seul élément du tuple. Pour les autres types de tuple, les attributs des classes correspondantes seront nommés de façon similaire (*class1* et *class2* pour *ClassXClassTuple*, *method1* et *attribute1* pour *MethodXAttributeTuple*).

6.2 Diagrammes

6.2.1 Les diagrammes dans MDP

Dans un système logiciel de grande taille, une métrique peut générer des milliers voire des dizaines de milliers de résultats. Pour rendre l'investigation de ces résultats plus commode, MDP utilise la visualisation d'information.

En fait, MDP ne fournit aucune visualisation concrète, il fournit l'infrastructure nécessaire pour permettre une intégration facile des métriques avec des diagrammes. MDP se veut capable d'intégrer n'importe quel diagramme venant de n'importe quelle composante tiers-partie. C'est pourquoi MDP impose très peu de restrictions sur le genre de diagrammes qu'il peut supporter. Concrètement, dans MDP, un diagramme c'est une façon quelconque de représenter graphiquement un ensemble de *ModelElement*s de un ou plusieurs types pré-déterminés.

6.2.2 Points de liberté

De plus, un diagramme possède un ou plusieurs *points de liberté* permettant l'ajout de résultats de métriques au diagramme. Un point de liberté est un aspect du diagramme (taille d'un glyph, position d'un glyph, couleur d'un glyph, etc) i.e. une variable visuelle, qu'on peut faire varier en fonction d'une valeur externe. Tout comme les métriques, un point de liberté est associé à un tuple de méta-éléments. *Ce tuple nous révèle les types des ModelElements qui sont visuellement associés au point de liberté et, par conséquent, il nous indique le type de métrique auquel le point de liberté est habilité à être associé pour faire la visualisation des résultats des métriques.* La Définition 15 définit le concept de point de liberté.

Définition 15 : Point de liberté

Un point de liberté est un triplet (A,B,C) tel que A est un tuple de méta-élément qui nous révèle à quels types de *ModelElements* le point de liberté est visuellement associé, B une chaîne de caractères qui fait partie de l'ensemble

{"position", "size", "texture ", "color" , "orientation", "curvature", "shape", "other"}

dans lequel chaque élément désigne une variable visuelle, à l'exception de *other* qui signifie que la variable visuelle n'est pas l'une de celles dans l'ensemble, et C est un entier qui représente le nombre de valeurs différentes que la dimension peut prendre ou, le mot « continuous » qui signifie que la dimension est continue. Comme pour les métriques (voir Définition 11) le *degré* d'un point de liberté est égal à la cardinalité de son type de tuple.

6.2.3 Règles d'association entre les métriques et les points de liberté

Le but premier de MDP est de permettre l'intégration de métriques avec les différents points de liberté d'un diagramme pour faire la visualisation des résultats des métriques. Bien sûr, les différents points de liberté fournis par les diagrammes ne sont pas également adéquats pour faire la visualisation des résultats d'une certaine métrique. C'est pourquoi MDP fournit un ensemble de règles d'associations entre les points de liberté et les métriques qui optimisent l'efficacité des visualisations. Ces règles d'associations sont basées sur les concepts de type de métrique (Définition 12) et de point de liberté (Définition 15) définis précédemment. Pour qu'une métrique puissent être visuellement associé avec un point de liberté il faut que :

1) Leur type de tuples soit compatible

2) La variable visuelle du point de liberté soit compatible avec le type d'échelle de la métrique.

Compatibilité des types de tuples

Pour que les résultats des métriques soient présentés dans le contexte des *ModelElement*s à partir desquels ils ont été calculés, ils doivent être visuellement associés à ces mêmes *ModelElement*s dans les diagrammes. Il semble donc, à priori, que le type du tuple d'une métrique doit être le même que le type du tuple du point de liberté pour que la métrique soit visualisable avec le point de liberté. Cependant, une certaine flexibilité peut être permise. La Définition 16 nous aidera à définir cette flexibilité.

Définition 16 : Super-type d'un type de tuple

> Soit le type de tuple **[a,b]** et le type de tuple **[x,y]** où **a**, **b**, **x** et **y** sont des méta-éléments dans un certain méta-modèle **MM**. Le type de tuple **[a,b]** est un super-type du type de tuple **[x,y]** si et seulement si le méta-élément **a** est un super-type du méta-élément **x** ou si le méta-élément **b** est un super-type du méta-élément **y** ou les deux.

Ici deux questions se posent. Premièrement, si le type du tuple du point de liberté est un super-type du type du tuple de la métrique, l'association est-elle possible? Ici la réponse est non. Pour démontrer cela, supposons qu'une métrique a comme type de tuple **[Méta-élément2]** et qu'un point de liberté a **[Méta-élément1]** (voir Figure 14). Le fait qu'un des points de liberté d'un diagramme ait **[Méta-élément1]** comme type de tuple signifie que le diagramme peut servir à représenter des instances de Méta-élément3. Or une métrique qui s'applique sur le Méta-élément2 ne s'applique pas nécessairement sur le Méta-élément3, ce qui pourrait causer des incohérences au niveau de la visualisation.

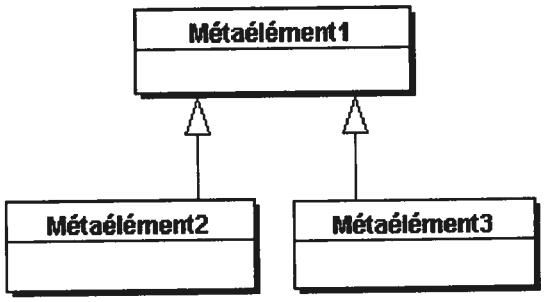


Figure 14: Trois méta-éléments quelconque

Maintenant, si le type du tuple de la métrique est un super-type du type du tuple du point de liberté, l'association est-elle possible? Dans ce cas la réponse est oui car une

métrique qui s'applique sur le type de tuple **[Méta-élément1]** s'applique forcément sur le type de tuple **[Méta-élément2]**. L'association est donc possible.

En résumé, une métrique est compatible avec un point de liberté si le type du tuple de la métrique est le même ou un super-type du type du tuple du point de liberté.

Compatibilité entre les variables visuelles et les types d'échelles

Les différents aspects de la théorie de la mesure et de la visualisation d'information présentés dans cette section nous permettent de déduire quelles sont les variables visuelles les plus appropriées pour chaque type d'échelles.

Le Tableau 10 donne un indice d'adéquation de 0 à 3 à chaque couple variable/échelle. L'indice 0 est donnée si l'utilisation de la variable est impossible ou extrêmement inappropriée pour le type d'échelles. L'indice 1 est donné si l'utilisation de la variable est possible mais que la variable ne permet pas de représenter correctement certaines caractéristiques du type d'échelles ou encore donne faussement l'impression que le type d'échelles possède ces caractéristiques. L'indice 2 est donné si l'utilisation de la variable est appropriée mais cause certains problèmes moins importants et l'indice 3 est donné si l'utilisation de la variable est appropriée. Les informations contenues dans ce tableau sont un amalgame des avis des différents auteurs mentionnés dans la section 2.2 mais sont surtout basée sur les informations du Tableau 5.

	Plan X-Y	Taille	Texture-Grain	Couleur	Orientation	Courbure	Forme
Nominal	2	1	3	3	2	1	3
Ordinal	3	3	3	3	1	2	1
Intervalle	3	3	2	2	1	1	0
Ratio	3	3	2	2	1	1	0
Absolu	3	3	2	2	1	1	0

Tableau 10: Indice d'adéquation des variables visuelles vis-à-vis les types d'échelles

Plan X-Y. Comme l'explique Bertin, les deux dimensions du plan sont appropriées pour tous les types de tâches (association, sélection, ordre, quantité) et donc à tous les types d'échelles. Nous mettons cependant un bémol sur leur utilisation pour les

échelles de type nominal car celles-ci ne sont pas ordonnées et que les dimensions du plan ont tendance à être associées avec des échelles ordonnées.

Taille. De la même façon, les variables visuelles de type taille sont appropriées pour toutes les échelles de type ordonné mais pas pour les échelles de type nominal car la taille n'est pas une variable visuelle associative (voir Tableau 5).

Texture-Grain. Bertin attribue des aptitudes pour l'association, la sélection, et l'ordre ce qui implique les types d'échelles nominal et ordinal mais pas les types d'échelles quantitatifs soit intervalle, ratio et absolue. Healey[40] de son coté croit pouvoir codé jusqu'à trois dimension ordinal dans la texture mais les résultats de ses expériences ne sont pas très concluantes. Nous dirons donc que la texture peut fournir une dimension aux échelles de type nominal et ordinal. Pour les types d'échelle quantitatifs (intervalle, ratio, absolue) l'avis général semble être que l'utilisation de la texture est possible mais pas très appropriée.

Couleur. Ici on rencontre une cassure entre la terminologie utilisée par Bertin[7] et les autres auteurs. Bertin parle de la *valeur* et de la *couleur* comme de deux variables visuelles différentes. La valeur est une variable visuelle continue constituée des différentes couleurs présentes sur un axe qui traverse l'espace de couleur. Bertin utilise les tons de gris mais tout autre axe peut servir. Il voit cette variable visuelle apte à des tâches de sélection et d'ordre, ce qui n'entraîne que les échelles de type ordinal. Pour la variable de la couleur, qui pour Bertin signifie les couleurs pures, les seuls types de tâches sont l'association et la sélection et donc, les échelles de type nominal seulement.

Donc en utilisant la variable visuelle de la couleur de différentes façons on peut l'adapter à des échelles de type nominal or ordinal.

D'autres auteurs croient la variable de la couleur appropriée non seulement pour les tâches d'ordre mais aussi pour les tâches quantitatives. Ware va encore plus loin et utilise la couleur pour coder trois dimensions quantitatives [85]! Mais dans ce même ouvrage, il prend bien soin de décrire plusieurs inconvénients à l'utilisation de cette

technique. Nous croyons cependant que la couleur peut être utilisé pour coder *une* dimension quantitative assez bien (indice de 2) en choisissant un axe dans l'espace de couleur.

Orientation. Bien que Treisman ait établi que différentes orientations peuvent donner lieu à des processus pré-attentifs, cette variable ne semble appropriée que pour les échelles de type nominal. De plus, sa longueur de seulement quatre constitue un inconvénient qui empêche l'attribution de l'indice 3 même pour ce type d'échelles.

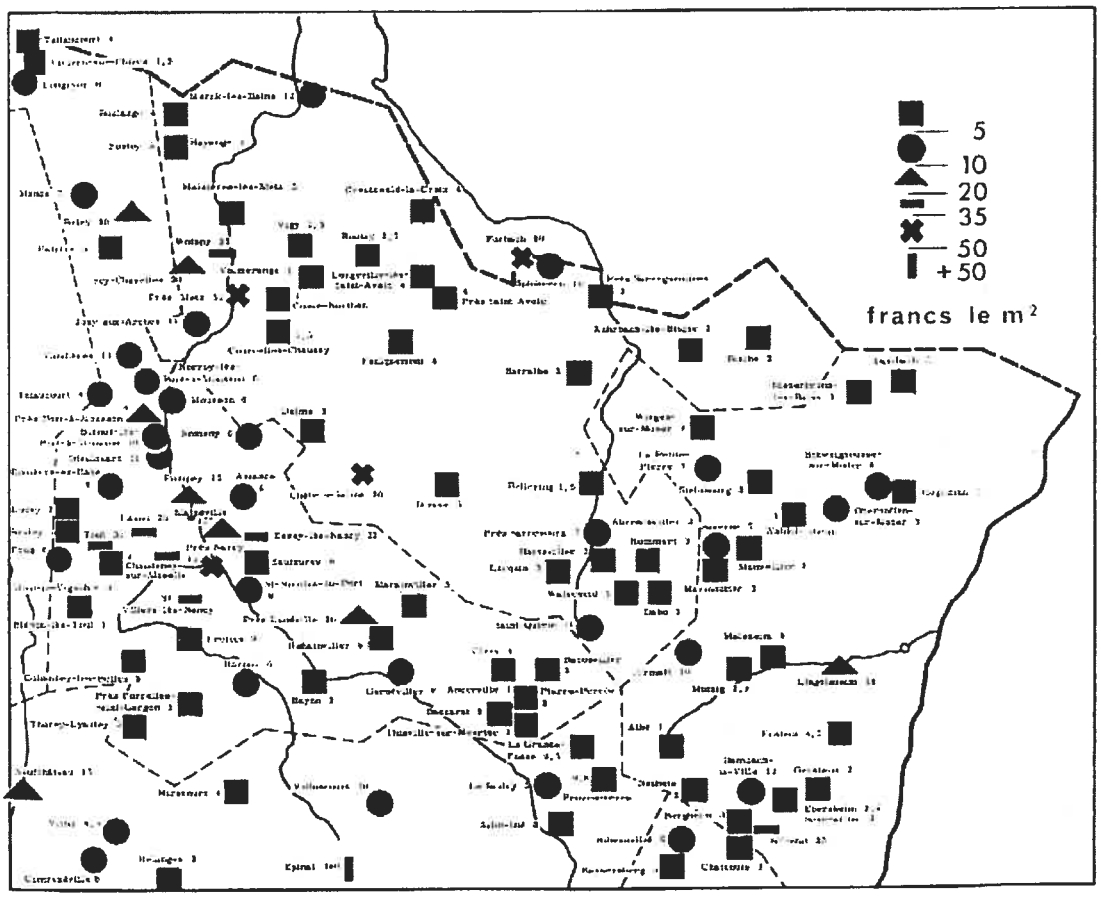
Courbures. La courbure d'une ligne est la seule variable visuelle du Tableau 10 qui n'est pas étudié par Bertin. Cependant, on peut facilement déduire que cette variable visuelle n'est pas associative, qu'elle est sélective, ordonnée mais pas quantitative. Le seul type d'échelles approprié est donc le type d'échelles nominal mais étant donné la très courte longueur de cette variable (trois), on ne peut attribuer que l'indice 2.

Forme. La variable visuelle de la forme est très appropriée pour les tâches associatives et donc pour les échelles de type nominal. Pour les échelles de type ordinal, cette variable est plutôt inappropriée (voir Figure 15).

La Figure 15 et la Figure 16 sont tirées de l'ouvrage de Bertin, et elles constituent un bon exemple de ce qui se produit lorsqu'une variable visuelle inappropriée pour l'ensemble de données visualisé est utilisée. Les deux figures visualisent un ensemble de données sur une échelle ordinale. La visualisation de la Figure 15 utilise la variable visuelle *forme* qui a un indice de 1 dans le Tableau 10 alors que la Figure 16 utilise la variable visuelle *taille* qui a un indice de 3 dans le Tableau 10. Dans la première figure, l'analyse des données nécessite un va-et-vient constant de l'attention entre la légende et la figure elle-même. En particulier, on ne peut pas répondre facilement à la question « Dans quelle région sont les terrains les plus chers ? ». Avec la visualisation de la deuxième figure, la réponse à cette question est trouvée en une fraction de seconde.

Les données du Tableau 10 sont intégrées dans MDP pour permettre la construction de visualisation plus efficace. Au moment de l'intégration des métriques, un outil

basé sur MDP pourra utiliser ces données pour suggérer une utilisation optimale des différents points de liberté d'un diagramme et pour interdire d'autres utilisations incohérentes.



PRIX DU TERRAIN DANS LA FRANCE DE L'EST d'après l'hebdomadaire "ELLE" . Paris 1959

Figure 15 : Utilisation d'une variable visuelle de type *forme* pour des données sur une échelle de type ordinal

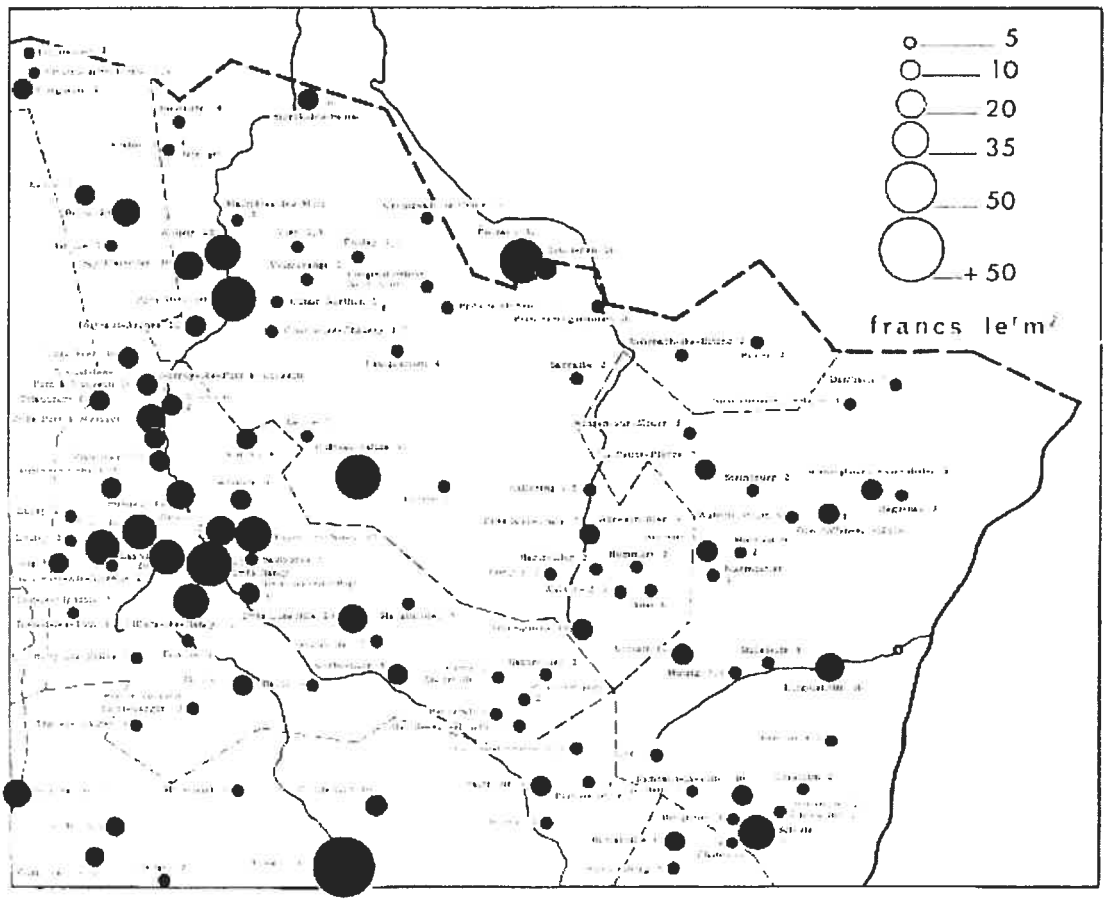


Figure 16: Utilisation d'un variable visuelle de type *taille* pour des données sur une échelle de type ordinal

6.2.4 Exemple simple d'un diagramme dans MDP

Diagramme structurel statique avec sept points de liberté

Il est maintenant temps de donner un exemple simple de ce à quoi un diagramme dans MDP peut ressembler. Sans doute, le diagramme le plus utilisé dans les outils de génie logiciel orienté objet est le diagramme structurel statique aussi appelé diagramme de classes. Comme l'a démontré Lanza [21][51], il existe de multiples façons d'intégrer des résultats de métriques dans un diagramme structurel statique. La Figure 17 donne un aperçu de ces possibilités.

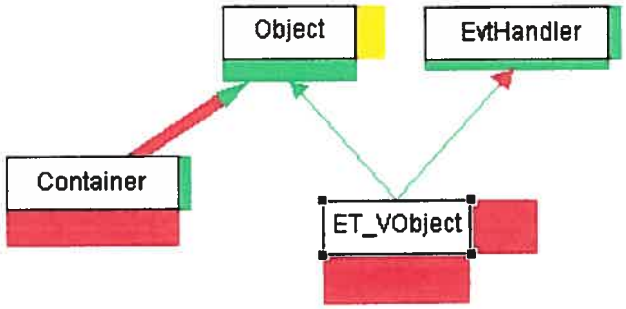


Figure 17: Exemple d'un diagramme dans MDP

Dans cette figure, les résultats de pas moins de sept métriques sont visualisés à l'aide des sept points de liberté du diagramme. Ces points de liberté sont énumérés dans le Tableau 11 avec le type de métrique qu'ils peuvent servir à visualiser.

Description du point de liberté	Type du tuple	Variable visuelle	Longueur	Spécification	Type d'échelles approprié (voir Tableau 10)
Largeur boîte coté droit	**[Class]**	Taille	Continue	**([Class], size, continuous)**	Ordinal, Intervalle, Ratio, Absolue
Couleur boîte coté droit	**[Class]**	Couleur	Continue	**([Class], color, continuous)**	Nominal, Ordinal
Hauteur boîte dessous	**[Class]**	Taille	Continue	**([Class], size, continuous)**	Ordinal, Intervalle, Ratio, Absolue
Couleur boîte dessous	**[Class]**	Couleur	Continue	**([Class], color, continuous)**	Nominal, Ordinal
Largeur corps de flèche	**[Class, Class]**	Taille	Continue	**([Class,Class], size, continuous)**	Ordinal, Intervalle, Ratio, Absolue
Couleur corps de flèche	**[Class, Class]**	Couleur	Continue	**([Class,Class], color, continuous)**	Nominal, Ordinal
Couleur tête de flèche	**[Class, Class]**	Couleur	Continue	**([Class,Class], color, continuous)**	Nominal, Ordinal

Tableau 11: Points de liberté du diagramme de la Figure 17

Tout d'abord, on remarque les deux boîtes de couleur situées en-dessous et sur le coté droit des classes de l'arbre d'héritage. Chacune de ces boîtes a une taille et une couleur, elles transportent donc les résultats de deux métriques chacune soit quatre résultats de métrique par nœud. On remarque aussi que ces boîtes sont *visuellement* associées à un seul *ModelElement* de type *Class*. Le type du tuple de chacun de ces quatre points de liberté est donc **[Class]**. Viennent ensuite trois autres points de liberté qui sont visuellement associés à un couple de *ModelElement* de type *Class* (**[Class,Class]**). Les résultats des métriques qu'ils visualisent se doivent donc eux aussi d'être associés à un couple de *ModelElement* de type *Class* ou d'un super-type de **[Class,Class]**.

On remarque que la Figure 17 modélise un système ou l'héritage multiple est permis. Cela nous force à identifier le type des tuples des trois derniers points de liberté comme étant **[Class,Class]**. Dans un diagramme structurel statique où l'héritage multiple n'aurait pas été permis, on aurait pu identifier les trois types de tuples comme étant **[Class]** (la classe au bas de chaque lien) car, dans un langage de programmation O-O à héritage simple, la super-classe d'une classe peut être identifiée de façon unique. Le produit cartésien n'est donc pas nécessaire. Mais puisque l'héritage multiple est permis, le type de tuple **[Class,Class]** est nécessaire. Maintenant si le méta-modèle utilisé modélise explicitement les relations d'héritage avec un méta-élément, comme c'est le cas dans UML avec le méta-élément *Generalization* alors le type des tuples des trois derniers points de liberté est tout simplement **[Generalization]**.

On remarque aussi que toutes les dimensions fournies sont continues. Pour les points de liberté qui ont *couleur* comme variable visuelle cela implique qu'une infinité de choix existent pour représenter des valeurs sur une échelle de type nominal ou ordinal. Par exemple, pour une métrique dont l'échelle est de type ordinal et comprends trois éléments, on pourrait choisir de les représenter avec les trois couleur blanc, gris et noir, ou encore orange, jaunes et rouge. Mais MDP, en prenant en considération le nombre d'éléments dans l'échelle, le type de l'échelle et les principes psychophysique décrit dans la section 2.2.1, propose des solutions optimales au

moment de l'intégration de la métrique dans un diagramme. Par exemple, dans la Figure 17, la taille des boîtes sur le coté et en-dessous des classes peut être utilisée pour visualiser le nombre d'attributs et de méthodes de la classe. Leur couleur peut être utilisé pour exprimer un niveau d'alerte du genre {« normal », « trop », « beaucoup trop »}. Puisque cette échelle ordinal comporte trois éléments, un choix évident pour représenter des résultats sur cette échelle avec la couleur est l'ensemble de couleur (vert, jaune, rouge) car cet ensemble a le même nombre d'élément que l'échelle est possède un ordre universellement reconnu. MDP proposera donc cet ensemble de couleur dans une situation semblable. Pour une métrique avec une échelle de type nominal et composé de sept élément, MDP proposera plutôt les sept couleur de Healey[40].

Scénario d'utilisation de MDP

Cette section a pour but d'exposer de quelle façon MDP facilite la visualisation des résultats des métriques. Pour bien comprendre le rôle de MDP il faut premièrement remarquer que la composante logicielle avec laquelle la visualisation de la Figure 17 a été générée pourrait avoir plusieurs origines complètement indépendantes de MDP.

Cette composante pourrait être une simple librairie qui permet de dessiner des arbres avec des nœuds carrés aux côtés plus ou moins épais et de différentes couleurs, avec une chaîne de caractères au milieu de ces noeuds et des fléches de couleur et de tailles différentes entre ces nœuds. Donc, une composante qui n'a rien à voir avec la visualisation de logiciels. Elle pourrait aussi provenir d'un outil très puissant qui permet la réutilisation de ses diagrammes grâce à une interface native. Finalement, cette composante pourrait avoir été entièrement créée à l'aide d'un engin à dessin (*drawing framework*).

Puisque tous ces types de composantes ont des interfaces non seulement différentes mais aussi de différentes natures (interface native, interface Java, XML, etc ...), MDP ne peut tout prévoir et, en général une certaine quantité de code devra être écrite pour adapter un diagramme à un outil basé sur MDP. Cependant MDP, en utilisant le patron de conception *Adapter* (voir [37] p.139), permet de concentrer toutes les

adaptations nécessaires dans une seule classe comme le montre la Figure 18. Dans cette figure, la composante de visualisation externe est représentée comme une classe normale (DiagrammeX) mais cette composante peut vraiment être de n'importe quelle nature. C'est le rôle de la classe *AdapteurX*, d'adapter le diagramme *DiagrammeX* à MDP. Ceci se fait en sous-classant la classe DiagramModel de MDP. Cette classe contient nombre de méthodes abstraites qui doivent être définies dans les sous-classes comme la classe *AdapteurX*. Entre autres, DiagramModel veut savoir

- Quel type de méta-éléments le diagramme peut contenir
- Quels sont les points de liberté du diagramme.

Il est de la responsabilité de la classe *AdapteurX* de décrire ces caractéristiques du diagramme. MDP se servira de ces informations pour vérifier si une association (point de liberté/métrique) est souhaitable ou même si elle est possible. C'est aussi la classe *AdapteurX* qui s'occupera de faire afficher les *ModelElements* et les résultats des métriques à la demande de MDP.

Une fois cette classe créée et testée, l'utilisateur de l'outil pourra, *à l'exécution*, intégrer des métriques avec le diagramme. C'est par ce processus d'essai et erreur que des visualisations efficaces seront découvertes.

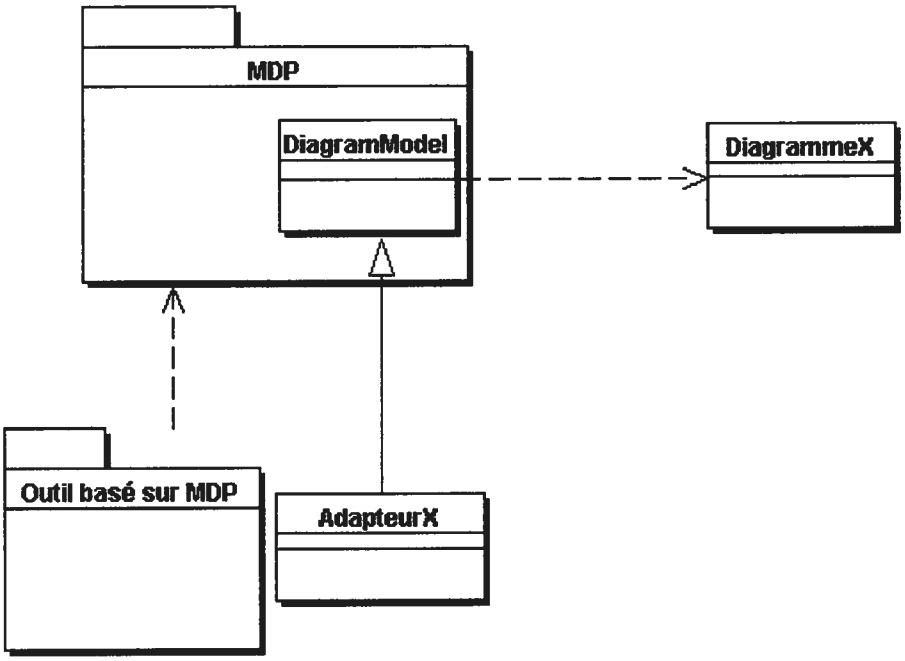


Figure 18: Principe général de MDP

6.3 Perspectives

Souvent, un seul diagramme ne sera pas suffisant pour se procurer toute l'information nécessaire à une certaine investigation. MDP reconnaît ce fait en permettant l'intégration de plusieurs diagrammes dans une même fenêtre et ce, même si ces diagrammes ne viennent pas de la même composante tiers-partie. On nomme une telle combinaison de diagrammes une *perspective*. Souvent les différentes visualisations présentes dans la perspective contiendront les mêmes *ModelElement*s et seront utilisées pour visualiser les mêmes métriques mais, d'une façon complémentaire. Par exemple, une visualisation comme celle de la Figure 17 qui peut faire l'investigation du nombre de méthodes et d'attributs par classe dans le contexte d'un arbre d'héritage serait bien complémentée par une courbe de densité qui donnerait immédiatement une idée de la distribution du nombre de méthode dans les classes du système.

Plusieurs autres fonctionnalités sont fournies par le mécanisme de perspectives de MDP, comme l'optimisation du calcul des métriques et la synchronisation du focus entre les diagrammes. Ces fonctionnalités seront décrites en détail au Chapitre 7.

Chapitre 7 : Le cadre d'application MDP

Ce chapitre décrit le cadre d'application MDP au niveau de sa conception. Il reprend la structure du chapitre précédent en séparant les classes de MDP en trois catégories : celles qui ont trait aux métriques, celles qui ont trait aux diagrammes et celles qui ont trait aux perspectives.

7.1 Métriques

7.1.1 Calcul des métriques

MDP permet de faire le calcul de métriques d'après la Définition 14. À l'aide d'un diagramme de classes UML, la Figure 19 expose les principales classes permettant de faire le calcul des métriques. Les différentes classes que l'on retrouve dans cette figure sont décrites ci bas.

Au moment du démarrage d'un outil basé sur MDP, un ensemble de définitions de métriques contenues dans un ou plusieurs fichiers XML sont chargées.

Metric. La classe *Metric* est la première instanciée lorsqu'une métrique est chargée. Les champs *name*, *shortName* et *description* sont créés à partir du contenu des balises <name>, <shortName> et <description> du fichier de définitions de métriques. Comme expliqué dans la Définition 13, une métrique peut avoir des contraintes qui limitent sont applicabilité. Ces contraintes sont définies en langage OCL à l'intérieur de la balise <pre-condition> du fichier de définitions de métriques. Ces contraintes se retrouvent sous la forme d'instances de la classe *OCL_Expression* et restent associées

à l'instance de la classe *Metric*. Lorsque la métrique est appliquée sur un tuple de *ModelElements* par l'intermédiaire de la méthode

+compute(Tuple aTuple) :Value,

les contraintes sont tout d'abord appliquées et, seulement si elles sont satisfaites, le tuple sera passé à une instance de la classe *Scale* pour que la métrique soit calculée sur celle-ci. L'instance de *Scale* renvoie le résultat du calcul sous la forme d'une instance de la classe *Value*, résultat qui est ensuite renvoyé à l'appelant de la méthode *compute*.

Scale. À chaque instance de la classe *Metric* est associé une instance de la classe *Scale* qui représente une échelle au sens de la théorie de la mesure. Conformément à la Définition 5 et à la Définition 14, une instance de la classe *Scale* possède un attribut de type *java.lang.Class* qui est une référence vers la classe qui définit le type de tuple de la métrique, i.e. une référence vers une des sous-classes de la classe *Tuple*. Les différents types d'échelles sont modélisés à l'aide des différentes sous-classes concrètes de *Scale* soit *NominalScale*, *OrdinalScale*, *IntervalScale*, *RatioScale* et *AbsoluteScale*. L'attribut *metricDefinition* est une référence vers l'expression OCL qui calcule le résultat de la métrique elle-même. Cette expression est définie avec une autre instance de la classe *OCL_Expression* qui est créé à partir du contenu de la balise <metric> dans le fichier de définitions de métriques. Les sous-classes de *Scale* implémentent la méthode abstraite

+performMesure(aTuple : Tuple) : Value

qui s'occupe d'appliquer l'expression OCL attachée à l'attribut *metricDefinition* sur le tuple *aTuple* fourni. Cette méthode s'occupe aussi de convertir le résultat obtenu depuis le système de type de OCL vers le système de type de Java pour ensuite enrober le résultat dans une instance de la classe *Value*. Ce résultat est ensuite retourné à l'objet *Metric* appelant.

***NominalScale*, *OrdinalScale*, *IntervalScale*, *RatioScale* et *AbsoluteScale*.** Comme mentionné ci-haut, pour les échelles de type *intervalle*, *ratio* et *absolue*, les résultats

s'expriment toujours à l'aide du même ensemble mathématique, soit l'ensemble des nombres naturels pour les échelles de type *absolue* et l'ensemble des nombres rationnels pour les échelles de type *ratio* et *intervalle* (voir Tableau 9). Les méthodes *performMesure* des classes IntervalScale et RatioScale retourneront donc une instance de *RealNumberValue*.

Pour les types d'échelles représentés par les classes *NominalScale* et *OrdinalScale*, les résultats s'expriment à l'aide d'un ensemble de chaînes de caractères qui est potentiellement différent pour chaque métrique. Cet ensemble de chaînes de caractères est défini dans les fichiers de définitions de métriques. Pour les types d'échelles *nominal* et *ordinal* les résultats sont sous la forme d'une chaîne de caractères. La méthode *performMesure* de la classe *NominalScale* renverra une instance de la classe *ID_SetValue* et celle de *OrdinalScale* renverra une instance de la classe *IDO_SetValue*. Les fonctionnalités communes à ces deux types d'échelles sont abstraites dans la classe *DiscreteFiniteScale*.

DiscreteFiniteScale. Cette classe contient les fonctionnalités propres aux types d'échelles discrets et finis, c'est-à-dire les échelles de type *Nominal* or *Ordinal*. Elle possède un attribut de type *OrderedSet* qui contient les différentes chaînes de caractères qui constituent l'échelle. Elle fournit également une méthode qui renvoie la cardinalité de cet ensemble et une méthode qui permet d'obtenir la position d'un élément à partir de la chaîne de caractères et vice-versa.

Tuple. La classe *Tuple* définit un type de tuples de *ModelElements* abstrait. Les différents types de tuples concrets sont modélisés à l'aide de sous-classes de *Tuple* comme *ClassTuple*, *ClassXClassTuple* *MethodXClassTuple*. Ces classes ne contiennent que des méthodes d'accès aux attributs.

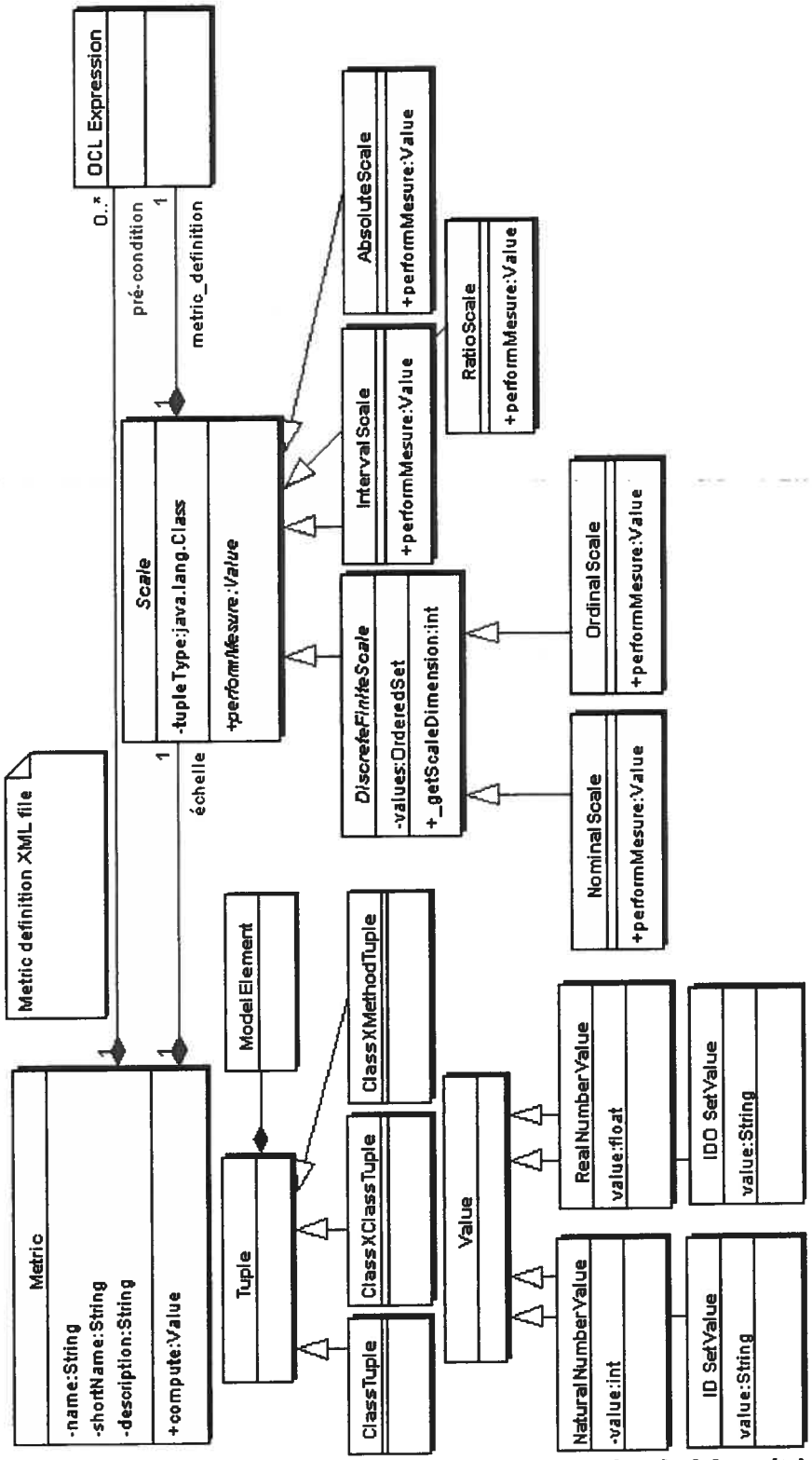


Figure 19: Le cadre d'application MDP: Classes permettant le calcul des métriques

Value. La classe *Value* modélise les résultats des métriques. Comme expliqué dans le Tableau 9 il y a quatre sortes de résultats : les naturels, les rationnels, les chaînes de

caractères faisant partie d'un ensemble ordonné et celles faisant partie d'un ensemble non-ordonné. Respectivement, les quatre classes *NaturalNumberValue*, *RealNumberValue*, *IDO_SetValue* et *ID_SetValue* permettent de contenir ces résultats.

7.1.2 Persistance des résultats des métriques

Les résultats des métriques calculées avec MDP peuvent être stockés dans le même dépôt que les *ModelElement*s. La Figure 20 illustre la structure d'objets qui rend cela possible.

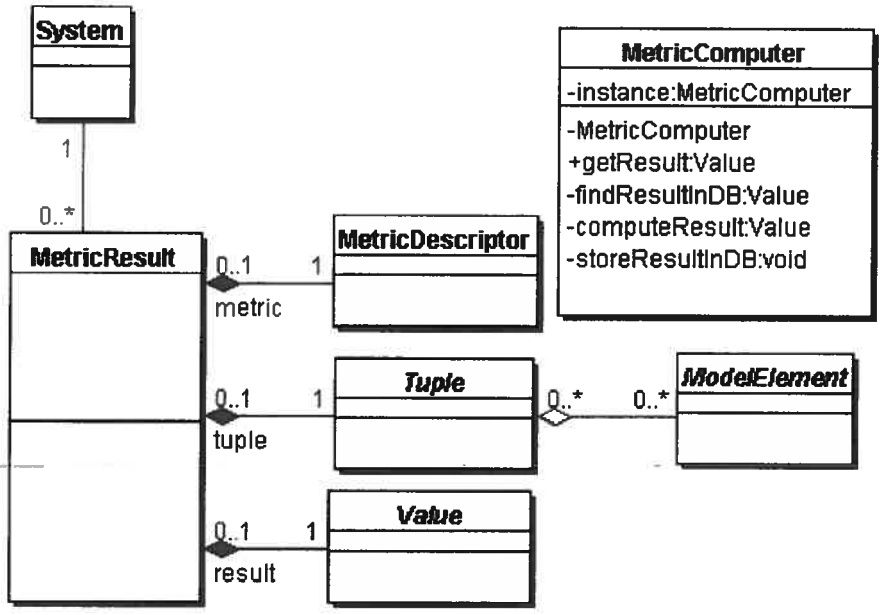


Figure 20: Structure de classes permettant la persistance des résultats des métriques

MetricResult. Le résultat du calcul d'une métrique sur un tuple de *ModelElement*s est modélisé avec la classe *MetricResult*. Les instances de cette classe conservent une référence vers un objet de type *Value* qui contient le résultat de la métrique proprement dit et une référence vers un objet de type *Tuple* qui contient le tuple de *ModelElement* à partir duquel la métrique à été calculée. Une référence vers un objet *MetricDescriptor* (voir Figure 25 ci-dessous) qui représente la métrique calculée est aussi conservée pour identifier l'origine du résultat de la métrique. Finalement, les objets *MetricResult* doivent être attachés à un objet déjà dans la base de données, objet représenté ici par la classe *System*, pour assurer leur persistance et pour permettre qu'ils soient récupérés facilement.

La recherche en fonction d'un résultat de métrique se fera tout d'abord en fonction de la métrique et ensuite en fonction du tuple pour lequel on désire avoir le résultat. C'est pourquoi, au niveau de la base de données, les objets *MetricResult* devraient être indexés premièrement par rapport à la métrique à laquelle ils sont associés et deuxièmement par rapport au tuple à partir duquel le résultat a été calculé. Cela améliorera grandement la performance d'un outil basé sur MDP.

MetricComputer. C'est la classe *MetricComputer* qui s'occupe de vérifier si le résultat d'une métrique est déjà dans la base de données. Cette classe est un singleton [37]. La méthode

getResult(aMetric: Metric, aTuple : Tuple) : Value

vérifie tout d'abord si le résultat demandé se trouve déjà dans le dépôt (méthode *findResultInDB*). Si il ne s'y trouve pas, la métrique *aMetric* est appliquée sur le tuple *aTuple* (la méthode *computeResult* appelle la méthode *compute* de la classe Metric), et le résultat est stocké dans la base de données avant d'être renvoyé.

7.2 Diagrammes

7.2.1 Intégration facile d'un nouveau diagramme dans MDP

DiagramModel. MDP permet d'ajouter facilement un nouveau diagramme aux outils qui sont basés sur lui. Pour ce faire, il suffit d'implanter un adaptateur sous la forme d'une sous-classe de *DiagramModel*. Cet adaptateur sera utilisé pour toutes les communications avec la composante logicielle qui réalise le diagramme (voir Figure 21). Cette adaptateur doit entre autres spécifier les types de *ModelElement*s qui peuvent être contenus dans le diagramme et, surtout, les points de liberté du diagramme.

Les méthodes suivantes doivent être implantées par les sous-classes de *DiagramModel*.

+initializeContentTypes( ) : java.lang.Class []

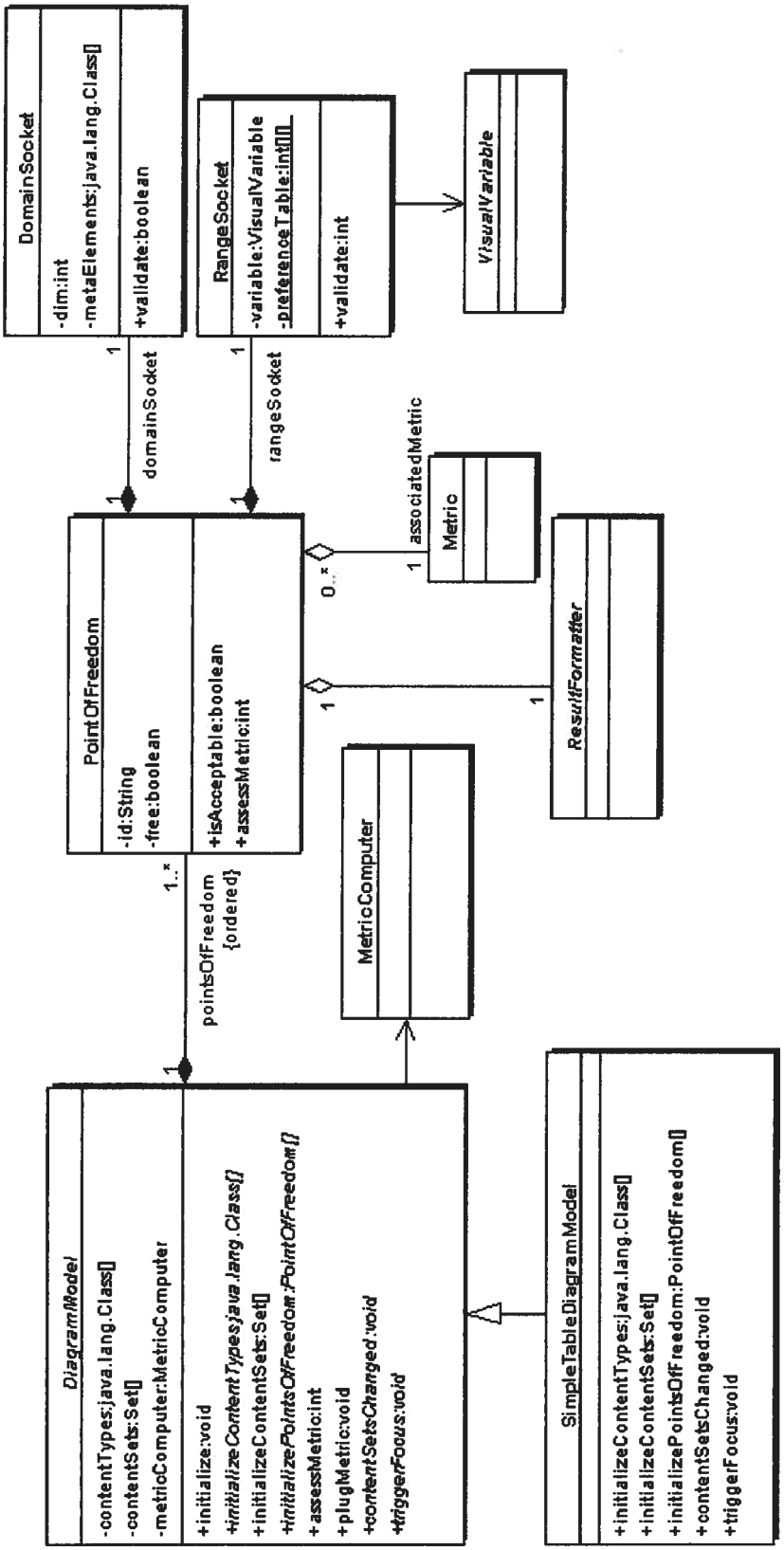


Figure 21: La cadre d'application MDP

Cette méthode doit retourner un tableau de références à des *java.lang.Class* qui sont les types de *ModelElement*s qui peuvent être contenus dans le diagramme. Ce tableau se retrouvera dans l'attribut *contentTypes* de *DiagramModel*. Un autre tableau de même longueur, *contentSets,* contient des références aux ensembles de *ModelElement*s qui sont contenus dans le diagramme. À la position *i* de ce tableau, on retrouve un ensemble de *ModelElement*s dont le type correspond à l'objet *java.lang.Class* se trouvant à la position *i* du tableau *contentTypes*.

+initializePointsOfFreedom( ) : PointsOfFreedom[]

Cette méthode doit retourner un tableau d'instances de la classe *PointsOfFreedom* qui décrivent les points de liberté du diagramme (voir Définition 15).

+contentSetsChanged( ): void

Cette méthode sera appelé par MDP chaque fois que le contenu du diagramme change. C'est la responsabilité de l'adaptateur de régénérer la visualisation à partir du contenu des *contentSets* à chaque fois que cette méthode est appelée.

+triggerFocus (aModelElementSet : Set) : void

Lorsque cette méthode est appelée, tous les éléments contenus dans l'ensemble *aModelElementSet* doivent apparaître avec un certain ornement qui attirera l'attention de l'utilisateur. Bien sûr tous les éléments de cet ensemble doivent faire partie de l'un des *contentSets* sinon une exception est lancée.

Il est important de noter que la classe *SimpleTableDiagramModel* de la Figure 21 ne fait pas partie de MDP. C'est l'exemple d'un adaptateur qui doit être implanté par l'utilisateur pour intégrer un certain diagramme du genre *spreadsheet* dans un outil basé sur MDP. Nous verrons au Chapitre 8 comment cette classe est implantée.

7.2.2 Logique d'intégration des métriques avec les diagrammes

DiagramModel. Les deux méthodes suivantes de la classe *DiagramModel* constituent l'interface accessible pour les outils basés sur MDP en ce qui concerne la logique d'intégration entre métriques et diagrammes. Premièrement la méthode

+assessMetric(aMetric : Metric, pof :PointOfFreedom) : int

renvoie l'indice d'adéquation de la métrique *aMetric* pour le point de liberté *pof* en se servant de la méthode *assessMetric* de la classe *PointOfFreedom*. C'est la méthode

+plugMetric(aMetric : Metric, pof : PointOfFreedom) : void

qui associe la métrique *aMetric* au point de liberté *pof* (met l'attribut *free* du point de liberté à *false*). Si l'association n'est pas acceptable, une exception est lancée.

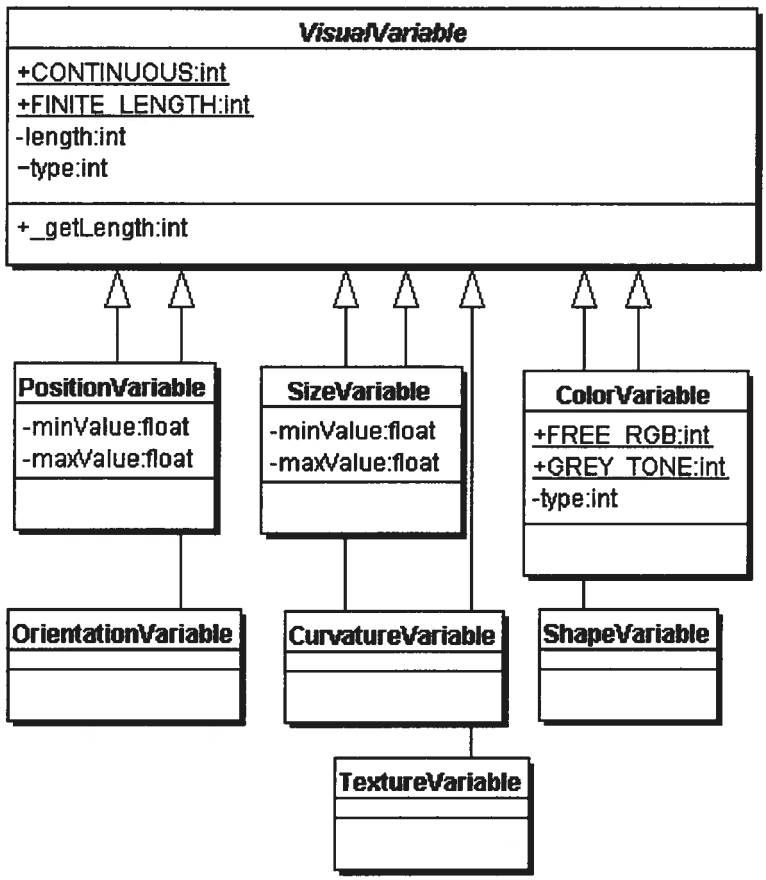


Figure 22: Hiérarchie de classes modélisant les différentes variables visuelles

PointsOfFreedom, DomainSocket, RangeSocket, VisualVariable. Un diagramme possède un ou plusieurs points de liberté qui sont modélisés par la classe *PointsOfFreedom*. Chaque instance de *PointsOfFreedom* est composé d'une instance de *DomainSocket* et d'une instance de *RangeSocket*. Un objet *DomainSocket* contient un tuple de références *java.lang.Class* qui définit le type de tuple du point de liberté. La méthode

+validate(Metric aMetric) : boolean

contient la logique par laquelle on peut décider si le type de tuple d'une métrique et le type de tuple d'un point de liberté sont compatibles (voir section 6.2.3, sous-section *Compatibilité des types de tuples*).

La classe *RangeSocket*, elle, contient une référence vers une instance de la classe *VisualVariable*. Cette hiérarchie de classes sert à décrire les détails de la variable visuelle (voir Figure 22) du point de liberté. Présentement, dans MDP, trois des sept variables visuelles peuvent être décrites avec des détails additionnels. Ces trois variables visuelles sont *position*, *taille* et *couleur* (voir Figure 22). Certains de ces détails permettent de faire un choix plus judicieux pour l'association d'une métrique alors que d'autres aident MDP à mieux choisir la valeur que prendra la variable visuelle pour un certain résultat de métrique.

Les classes *SizeVariable* et *PositionVariable* permettent de spécifier une valeur minimale et une valeur maximale que la variable visuelle peut prendre (attributs *minValue* et *MaxValue*) sous la forme d'un *float*. Si le nombre de position ou de taille possible est fini i.e. si la variable visuelle n'est pas continu, l'attribut *type* de la classe *VisualVariable* aura la valeur *VisualVariable.FINITE_LENGTH* et la longueur de la variable sera spécifiée avec l'attribut *length* de la classe *VisualVariable*.

La classe *ColorVariable* est plus complexe étant donné les multiples façons d'utiliser la variable visuelle *couleur*. Si la variable peut prendre toutes les couleurs du spectre de couleurs RGB, l'attribut *type* doit être à la valeur *ColorVariable.FREE_RGB*. Si la variable ne permet que les tons de gris, *type* doit être à la valeur

ColorVariable.GREY_TONE. Si la variable visuelle ne peut prendre que des couleurs discrètes comme valeur, c'est l'attribut *type* de la classe *VisualVariable* qui aura la valeur *VisualVariable.FINITE_LENGTH* de la même façon que pour les variables *position* et *taille*. Encore une fois, l'attribut *length* contiendra le nombre de couleurs différentes disponibles. Pour les autres variables visuelles, *orientation*, *courbure*, *texture* et *forme*, la seule chose qui peut être spécifiée c'est si la variable est une variable discrète ou continue (attribut *type* de la classe *VisualVariable* mis à la valeur *VisualVariable.FINITE_LENGTH ou VisualVariable.CONTINUOUS* respectivement) et, dans la cas ou la variable est discrète, on spécifie sa longueur avec l'attribut *length*.

La classe *RangeSocket* contient aussi la méthode

+validate(Metric aMetric) : int,

qui applique les données contenues dans le Tableau 10. Elle renvoie un indice d'adéquation de 0 à 3 en comparant l'attribut *variable,* qui est une référence vers la variable visuelle du point de liberté, et le type d'échelle de la métrique qui lui est fournie en paramètre. Si l'échelle de la métrique est type *Nominal* ou *Ordinal* et que le nombre total de valeurs dans l'échelle (obtenu avec la méthode *getScaleDimension( )* de la classe *DiscreteFiniteScale*) est supérieur à la longueur de la variable visuelle (attribut *length* de *VisualVariable*), l'indice 0 est renvoyé. Ceci ne peut se produire que pour les variables visuelles discrètes.

La classe *PointOfFreedom* possède également une chaîne de caractères (attribut *id*) qui identifie le point de liberté de façon unique à travers l'ensemble des points de liberté du diagramme. Un autre attribut de type booléen, *free*, indique si le point de liberté est libre où s'il est déjà occupé par une métrique. La méthode

+isAcceptable(aMetric : Metric) : boolean

renvoie *true* si il est *possible* d'utiliser le point de liberté pour la métrique donnée en paramètre. Plus précisément, cette méthode renvoie *true* si les types de tuple sont compatibles (information obtenue avec la méthode *validate* appelée sur l'attribut

domainSocket) et si l'indice d'adéquation pour le type d'échelle est plus grand que zéro (information obtenue en appelant la méthode *validate* sur l'attribut *rangeSocket*). Quant à la méthode

+assessMetric (aMetric : Metric) : int,

elle renvoie directement l'indice d'adéquation pour le type d'échelle de la métrique ou bien 0 si les types de tuples ne sont pas compatibles.

ResultFormatter. Une instance de la classe *ResultFormatter* est associée à chaque point de liberté. Cette classe permet de choisir une valeur appropriée pour la variable visuelle en fonction du résultat à visualiser tout en prenant en compte les détails fournis dans l'instance de *VisualVariable* associée au point de liberté. (voir Figure 23 pour plus de détails).

7.2.3 Aide au choix des valeurs (échelles quantitatives)

Un des obstacles majeurs à l'intégration des métriques avec les diagrammes est ce qu'on appelle le problème des bornes. On rencontre le problème des bornes pour les points de liberté qui sont associés à des métriques dont l'échelle est de type *intervalle*, *ratio* ou *absolue*. Ces métriques rendent des résultats sur l'échelle des nombres rationnels ou sur l'échelle des nombres naturels qui sont toutes deux infinies alors que les variables visuelles connues de MDP, elles, prennent des valeurs dans un intervalle fini. Puisque les résultats des métriques peuvent se trouver n'importe où sur une échelle infinie, il est impossible pour MDP de définir une projection fixe entre ces résultats et les valeurs que peuvent prendre les variables visuelles. Plusieurs solutions ont été explorées pour résoudre ce problème.

La solution la plus simple serait d'exiger de chaque métrique qu'elle fournisse un intervalle de valeurs à l'intérieur duquel la quasi-totalité de ses résultats se trouveraient. MDP pourrait alors établir une correspondance linéaire entre cet intervalle et l'intervalle des valeurs disponibles dans la variable visuelle. Dans l'éventualité où la métrique produirait des résultats en dehors de cet intervalle, ces

résultats serait représentés, selon le cas, par la valeur minimum ou maximum de la variable visuelle.

Cependant cette façon de faire est très peu appropriée et doit être rejetée car elle amènerait MDP à créer des visualisations extrêmement peu efficaces ou même inutiles. Par exemple, la distribution des résultats de la métrique NOMPC, *Number of Method per Class*, peut varier énormément d'un système à l'autre. On peut très bien imaginer un système où la quasi-totalité des classes ont entre 2 et 10 méthodes et un autre système ou la quasi-totalité des classes ont entre 20 et 100 méthodes. Maintenant si on attribue un intervalle de [1,25] à la métrique NOMPC, les résultats pour le premier système seront assez bien visualisés mais les résultats du deuxième système seront presque exclusivement visualisés à l'aide de la valeur maximum de la variable visuelle. Choisir un intervalle plus grand comme [1,150] assurerait que toutes les valeurs soient correctement représentées les unes par rapport aux autres mais, pour le premier système, elles seraient toutes concentrées dans le bas de l'intervalle que nous fournit la variable visuelle. Or dans MDP on veut bien sûr prendre avantage au maximum de l'intervalle de valeur disponible dans chaque variable visuelle pour provoquer des détections pré-attentives.

Les propos précédents montrent que MDP doit adapter la projection des résultats des métriques en fonction de chaque ensemble de résultats à visualiser. Lanza s'attaque au problème similaire mais plus restreint de la taille des nœuds d'un arbre [51]. Cet auteur commence par effectuer une projection linéaire des résultats des métriques vers la taille des nœuds en pixels (par exemple, un résultat de 5 visualisé avec un nœud de 5 pixels de haut). Si un des résultats de la métrique dépasse la taille maximale permise pour les nœuds, la taille de tous les nœuds est divisée par deux. On obtient donc une projection linéaire dont le rapport entre les différentes valeurs sera toujours correctement représenté.

Cette méthode semble appropriée pour MDP à priori mais certains ensembles de résultats particuliers peuvent encore avoir comme effet de générer des visualisations complètement inefficaces. Si on reprend le cas de la métrique NOMPC, on peut

s'imaginer un système où toutes les 300 classes ont entre 1 et 15 méthodes sauf une qui en a plus de 200. Ici, une projection linéaire entre 0 et 200 aura comme effet de visualiser plus de 99% des valeurs (299 classes sur 300) en utilisant seulement 1/12 de l'intervalle total de la variable visuelle. Cette visualisation nous permettra bien sûr d'identifier rapidement la classe potentiellement problématique qui contient 200 méthodes mais elle sera également peu utile pour analyser la distribution des 299 autres résultats puisqu'ils seront tous concentrés dans un petit intervalle de la variable visuelle. En plus, l'existence d'une classe aussi singulière serait probablement déjà connue de l'utilisateur.

MDP utilise des notions statistiques pour trouver un juste compromis. Le but ici c'est de trouver une façon de faire une projection linéaire qui, étant donné un ensemble de valeurs quelconque, définit un intervalle de valeurs à l'intérieur duquel la majorité des valeurs de l'ensemble se trouveront mais, sans qu'un petit groupe de valeurs extrêmes puisse avoir comme effet de compresser une grande partie des résultats à une extrémité de l'intervalle. Les limites de cet intervalle seront associées aux valeurs minimales et maximales de la variable visuelle. La méthode qui nous semble la plus appropriée est de prendre $[\mu - 2\sigma, \mu + 2\sigma]$, où μ est la moyenne arithmétique de tous les résultats de la métrique présente dans un diagramme et σ l'écart type. De cette façon si la distribution des valeurs est normale, 95 % des valeurs de la métriques seront à l'intérieur de l'intervalle et 5 % à l'extérieur.

Cette méthode comporte plusieurs avantages. Premièrement, elle assure que l'intervalle sera bien centré sur l'ensemble des résultats des métriques. Deuxièmement elle assure que l'intervalle sera assez large pour éviter que trop de valeurs ne soient représentées par le maximum où le minimum de l'intervalle et assez petit pour bien représenter la distribution des résultats. Bien sûr rien ne garantie que les distributions des résultats seront normales, mais même dans le cas d'une distribution très différente de la distribution normale, cette méthode donne des résultats acceptables. Par exemple, soit l'ensemble de résultats suivant :

[17,18,22,22,23,23,23,23,23,26,75,76,76,77,77,77,77,78,83,84].

Cette ensemble, loin de présenter une distribution normale, présente des valeurs très loin de la moyenne (50) et regroupées autour de deux pôles soit 23 et 77. Dans ce cas-ci, l'intervalle $[\mu - 2\sigma, \mu + 2\sigma]$ sera [21.1, 78.9], intervalle qui contient 16 des 20 valeurs. La variable visuelle utilisée représentera les valeurs 17 et 18 comme la valeur 21.1 et les valeurs 83 et 84 comme la valeur 78.9 ce qui représente une erreur acceptable étant donné que le but de le visualisation n'est pas nécessairement de donner une représentation exacte des données mais bien une représentation efficace.

La méthode comporte aussi quelques inconvénients. Dans un premier temps, la formule pour calculer l'écart type est quelque peu complexe et pourrait mener à une mauvaise performance pour les grands ensembles de résultats. Ensuite, puisque la projection entre les résultats des métriques et les variables visuelles est dépendante de l'ensemble de résultats lui-même, une même valeur sera représentée différemment pour des groupes de *ModelElement*s différents, ce qui empêchera les gains en efficacité qui auraient pu résulter d'une utilisation prolongée d'un même diagramme par un utilisateur.

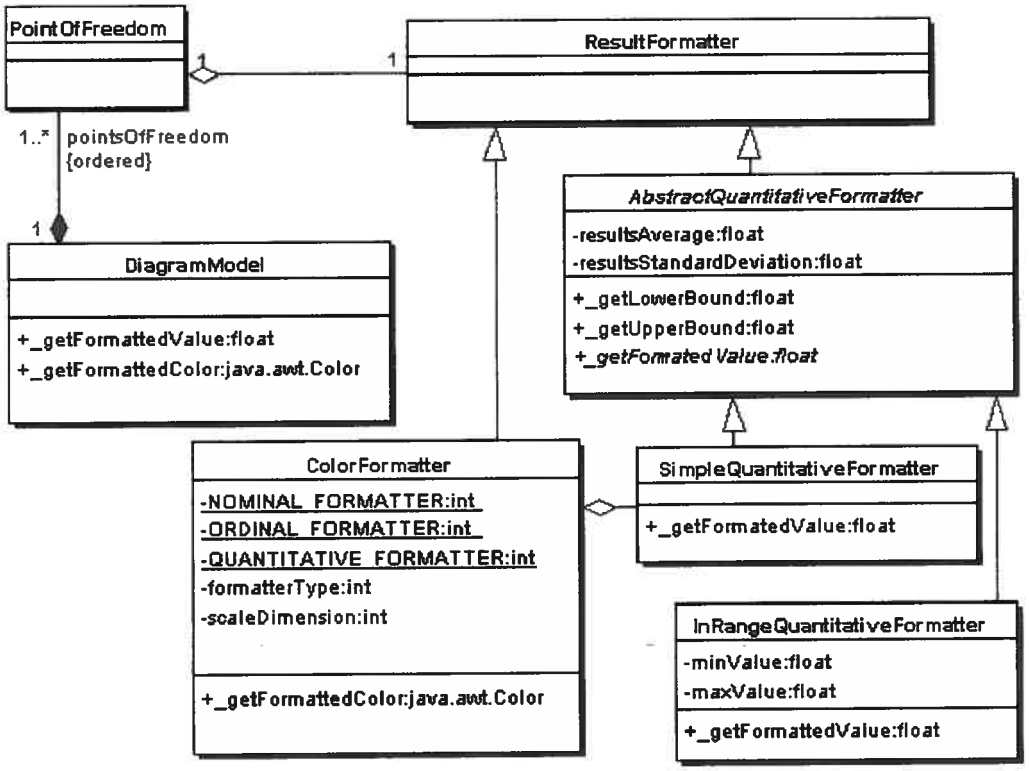


Figure 23: Hiérarchie de classes des *ResultFormatter*

AbstractQuantitativeFormatter. Dans le cadre d'application, ces fonctionnalités sont incluses dans la classe *AbstractQuantitativeFormatter* qui est une sous-classe de la classe *ResultFormatter* (voir figure 23). Les instances de cette classe maintiennent à jour les valeurs de deux attributs, *resultsAverage* et *resultsStandardDeviation* qui sont, respectivement, la moyenne et l'écart type des résultats de l'application de la métrique à tous les tuples de *ModelElement*s présents dans le diagramme. (voir Figure 23). La méthode *getLowerBound* renvoie $\mu - 2\sigma$ et *getUpperBound* renvoie $\mu + 2\sigma$. La méthode

getFormatedValue (aValue : Value) : float

est abstraite. Les sous-classes concrètes de *AbstractQuantitativeFormatter* implantent cette méthode et renvoient un *float* qui définit la valeur que doit prendre la variable visuelle. L'implantation de cette méthode doit prendre en compte les détails de la variable visuelle fournie par l'adaptateur du diagramme et qui sont contenus dans l'instance de *VisualVariable* associé au point de liberté.

InRangeQuantitativeFormatter. La classe *InRangeQuantitativeFormatter* est utilisée pour les variables visuelles *taille* et *position* qui ont une valeur minimale et maximale bien déterminée (attribut *minValue* et *maxValue* des classes *SizeVariable* et *PositionVariable*, voir Figure 22). Ici la méthode *getFormatedValue* fournit une projection linéaire entre l'intervalle fournie par les méthodes *getLowerBound* et *getUpperBound* et les limites de la variable visuelle spécifiées par les attributs *minValue* et *maxValue*.

SimpleQuantitativeFormatter. Cette classe est similaire à celle décrite dans le paragraphe précédent. Elle fait cependant abstraction des valeurs limites de la variable visuelle. Sa méthode *getFormatedValue* renvoie une valeur entre 0.0 et 1.0 qui correspond à la position de la valeur *aValue* dans l'intervalle fourni par *getLowerBound* et *getUpperBound*.

Puisque la méthode des deux écarts types ne s'applique que pour les types d'échelles quantitatifs, elle ne servira le plus souvent que pour trouver des valeurs pour les

variables visuelles *taille* et *position* qui ont des indices d'adéquation de 3 pour les types d'échelle quantitatif. Cependant, comme nous le verrons dans la prochaine section, elle peut aussi servir pour trouver des valeurs pour la variable visuelle *couleur* car elle a un indice de 2 (acceptable) pour les types d'échelles quantitatifs.

Pour bénéficier des fonctionnalités fournis par le *AbstractQuantitativeFormatter* associé à un certain point de liberté, un adaptateur appelle la méthode

getFormatedValue (pof : PointOfFreedom, aValue : Value) : float

de la classe *DiagramModel*.

7.2.4 Aide au choix des valeurs (échelles discrètes)

La section précédente explique comment MDP aide à choisir les valeurs pour les variables visuelles qui sont associé à une métrique dont l'échelle est de type quantitatif. MDP fournit également une certaine aide pour les métriques dont l'échelle est de type discret i.e. de type ordinal ou nominal et qui sont associés à la variable visuelle *couleur*. Cette aide est rendue disponible aux adaptateurs via la méthode

getFormattedColor(pof : PointOfFreedom, aValue: Value) : java.awt.Color

de la classe *DiagramModel*. Cette méthode appelle ensuite la méthode

getFormattedColor(aValue : Value) : java.awt.Color

de la classe *ColorFormatter* au travers du point de liberté *pof*.

Cette méthode contient la plupart des notions psychophysique reliées à la couleur qui ont été discutées dans le Chapitre 2. Elle prend en paramètre un résultat de métrique (*aValue*) et renvoie la couleur appropriée sous la forme d'un objet *java.awt.Color*. L'un des constructeurs de la classe *java.awt.Color* permet de spécifier une couleur à l'aide de trois valeurs de type *float* entre 0.0 et 1.0 (ex : [0.6, 0.7, 0.3]) selon le modèle RGB [35].

Une variable visuelle *couleur* peut se retrouver associée à une échelle de n'importe quel des cinq types (indice d'adéquation 3 pour *ordinal nominal*, indice 2 pour les trois autres). Dans le cas d'une échelle de type nominal, les septs couleurs de Healey [39] sont utilisées. Le Tableau 12 contient les valeurs des trois composantes RGB de chacune de ces couleurs. Il est à noter que ces valeurs sont approximatives car, de façon surprenante, l'ouvrage de Healey ne fournit pas ces sept couleurs précisément, seulement une méthode quelque peu floue pour les obtenir. De plus, calculer ces valeurs implique une conversion du modèle de couleur CIE LUV[35] vers le système de couleur RGB dans lequel la couleur résultante réelle dépend du moniteur utilisé.

Le Tableau 13 spécifie quelles couleurs sont utilisées pour chaque longueur d'échelle possible, de 2 à 7. Ces couleurs ont été choisies de façon à être aussi loin que possible dans l'espace de couleurs pour favoriser les processus de détection pré-attentifs.

Couleur	R	G	B
Vert	0.07	0.57	0.19
Jaune	0.44	0.47	0.01
Jaune-Rouge	0.76	0.37	0.22
Rouge	0.97	0.28	0.25
Rouge-Violet	0.88	0.27	0.60
Violet-Bleu	0.47	0.38	0.79
Bleu-Vert	0.07	0.53	0.57

Tableau 12: Composantes RGB approximatives des septs couleurs de Healey [39]

Dimension de l'échelle	Couleurs utilisées
2	Bleu-Vert, Jaune-Rouge
3	Bleu-Vert, Jaune, Rouge-Violet
4	Vert, Violet-Bleu, Rouge, Jaune
5	Vert, Violet-Bleu, Rouge, Jaune
6	Vert, Bleu-Vert, Violet-Bleu, Rouge, Jaune
7	Vert, Bleu-Vert, Violet-Bleu, Rouge-Violet, Rouge, Jaune

Tableau 13: Couleurs utilisées pour les différentes dimensions d'échelle de type nominal

Si plus de sept couleurs sont nécessaires, de nouvelles couleurs sont créés au milieu de chaque intervalle successivement selon le processus de Healey. Cependant, plusieurs résultats expérimentaux obtenus par Healey démontrent qu'au-delà de sept couleurs, les processus de détection deviennent de moins en moins pré-attentifs. Sept est le plus grand nombre pour lequel le temps de détection ne dépend pas ou peu du nombre d'éléments dans la visualisation.

Maintenant, dans le cas des échelles de type ordinal, les différentes valeurs des variables visuelles se doivent de posséder un ordre naturel pour que la visualisation soit efficace (voir Figure 15 et Figure 16). Or, il n'y a pas de façon universelle d'ordonner les couleurs mis à part l'ordre fourni par le spectre des couleurs (couleurs de l'arc-en-ciel). Cependant, quelques expériences rapides nous ont convaincus que cet ordre n'est pas reconnu facilement par tous les individus. Lá seule façon de construire un ensemble de couleurs d'une cardinalité quelconque et dont les éléments sont naturellement ordonnés semble être de faire varier la *luminescence* d'une couleur précise (luminescence au sens du modèle de couleur HLS [35]) du plus pâle au plus foncé (luminescence décroissante). Par exemple, si on choisit la couleur rouge (Hue=0°, Saturation=1.0), on peut faire varier la luminescence L entre 0.9 et 0.33 pour obtenir des rouges de plus en plus foncés. Pour L entre 1.0 et 0.9, la couleur serait trop semblable au blanc, et pour L entre 0.33 et 0 elle serait trop semblable au noir. En termes du modèle de couleur RGB, cette variation correspond à une variation linéaire entre [1.0, 0.8, 0.8] et [1.0, 0.0, 0.0] (les composantes G et B restent égales) et ensuite entre [1.0, 0.0, 0.0] et [0.67, 0.0, 0.0], ce qui est facile à implémenter. Les différentes valeurs utilisés sont choisies à intervalle régulier. La grandeur de ces intervalles dépend bien sûr de la dimension de l'échelle.

Couleur	Intervalle HSL	Intervalle RGB
Rouge	{0°, 1.0, 0.9}-{0°, 1.0, 0.33}	[1.0, 0.8, 0.8] – [1.0, 0.0, 0.0] – [0.67, 0.0, 0.0]
Jaune	{60°, 1.0, 0.9}-{60°, 1.0, 0.33}	[1.0, 1.0, 0.8] – [1.0, 1.0, 0.0] – [0.67, 0.67. 0.0]
Vert	{120°, 1.0, 0.9}-{120°, 1.0, 0.33}	[0.8, 1.0, 0.8] – [0.0, 1.0, 0.0] – [0.0, 0.67, 0.0]
Cyan	{180°, 1.0, 0.9}-{180°, 1.0, 0.33}	[0.8, 1.0, 1.0] – [0.0, 1.0, 1.0] – [0.0, 0.67, 0.67]
Bleu	{240°, 1.0, 0.9}-{240°, 1.0, 0.33}	[0.8, 0.8, 1.0] – [0.0, 0.0, 1.0] – [0.0, 0.0, 0.67]
Magenta	{300°, 1.0, 0.9}-{0°, 1.0, 0.33}	[1.0, 0.8, 1.0] – [1.0, 0.0, 1.0] – [0.67, 0.0, 0.67]

Tableau 14: Intervalle de couleur utilisé par MDP pour les échelles de type ordinal selon les modèles de couleur HSL et RGB [35]

Dimension	Couleurs utilisées					
4	[0.8,0.8,1.0]	[0.42,0.42,1.0]	[0.05,0.05,0.1]	[0.0,0.0,0.67]		
5	[0.8,0.8,1.0]	[0.52,0.52,1.0]	[0.23,0.23,1.0]	[0.0,0.0,0.95]	[0.0,0.0,0.67]	
6	[0.8,0.8,1.0]	[0.57,0.57,1.0]	[0.34,0.34,1.0]	[0.12,0.12,1.0]	[0.0,0.0,0.89]	[0.0,0.0,0.67]

Tableau 15: Exemple de choix de couleurs par MDP pour des échelles de type ordinal : couleur bleu avec dimension de 4,5 et 6. Les couleurs sont exprimées selon le modèle RGB

MDP fournit six couleurs différentes à partir desquels des ensembles de couleurs ordonnées peuvent être construits. Le choix de ces six couleurs peut être présenté à l'utilisateur d'un outil basé sur MDP au moment de l'intégration des métriques. Le Tableau 14 énumère ces couleurs ainsi que l'intervalle utilisé exprimé en fonction des modèles de couleur HLS et RGB. Le Tableau 15 montre quelles couleurs seront utilisées pour des échelles de type *ordinal* de grandeur 4, 5 et 6. La couleur bleue est prise en exemple.

Finalement, dans le cas des échelles de type quantitatifs, les intervalles fournis par le Tableau 14 seront encore une fois utilisées. Cependant, cette fois les résultats des métriques seront projetés dans le spectre continus de ces intervalles. Ici, puisqu'on a affaire aux types d'échelles continus, le problème des bornes discuté précédemment refait surface.

ColorFormatter. *ColorFormatter* est une sous-classe de *ResultFormatter* qui contient toutes les données décrites dans cette section. Son attribut *formatterType* peut prendre les valeurs *ColorFormatter.NOMINAL_FORMATTER*, *ColorFormatter.ORDINAL_FORMATTER* ou *ColorFormatter.QUANTITATIVE_FORMATTER* dépendamment du type d'échelle de la métrique à laquelle le point de liberté est associé. Dans les deux premiers cas, l'attribut *scaleDimension* contiendra la dimension de l'échelle de la métrique. Maintenant, pour les échelles de type *ordinal* ou *quantitatif*, l'attribut *choosedColor* de type *java.awt.Color* réfèrera à l'une des six couleurs de base fournies par MDP. À l'aide de ces trois attributs,

formatterType, *scaleDimension* et *choosedColor* la méthode *getFormattedColor* peut renvoyer la bonne couleur (spécifier par le Tableau 12, le Tableau 13 et le Tableau 14) à la méthode *getFormattedColor* de la classe *DiagramModel*.

Pour les variables visuelles *couleur* associées à une métrique dont le type d'échelle est quantitatif, le problème des bornes discuté précédemment ressurgi. C'est pourquoi les instances de *ColorFormatter* se serviront d'une instance de *SimpleQuantitativeFormatter* pour trouver la projection d'un résultat quantitatif vers les intervalles du Tableau 14.

Ici, il faut noter que ce n'est pas du ressort des adaptateurs de créer les instances de *ResultFormatter*, c'est la tâche de MDP. En général, c'est au moment de l'intégration d'une métrique avec un point de liberté que le *ResultFormatter* sera créé par MDP avec des paramètres venant de l'échelle de la métrique et de la variable visuelle du point de liberté.

7.3 Perspectives

7.3.1 Mécanisme de présentation des perspectives

PerspectiveDiagram. Dans MDP, une perspective est composée de plusieurs diagrammes concrets, i.e. des diagrammes avec des métriques intégrées. Chaque diagramme occupe un endroit précis dans la perspective. Une perspective peut compter jusqu'à 9 diagrammes qui sont présentés dans une même fenêtre. Bien que MDP ne fournisse aucun diagramme, il fournit les fonctionnalités nécessaires pour intégrer plusieurs diagrammes dans une même fenêtre et ainsi créer des perspectives.

La Figure 30 donne un exemple de ce à quoi peut ressembler une perspective. Chaque case de la perspective peut être redimensionnée à la guise de l'utilisateur. À la base, une perspective contient 3 rangées et 3 colonnes et donc, 9 cases. Mais un diagramme peut occuper plusieurs cases à la fois (1x2, 1x3, 2x2, etc.). La Figure 24 illustre les classes qui permettent de réaliser les perspectives.

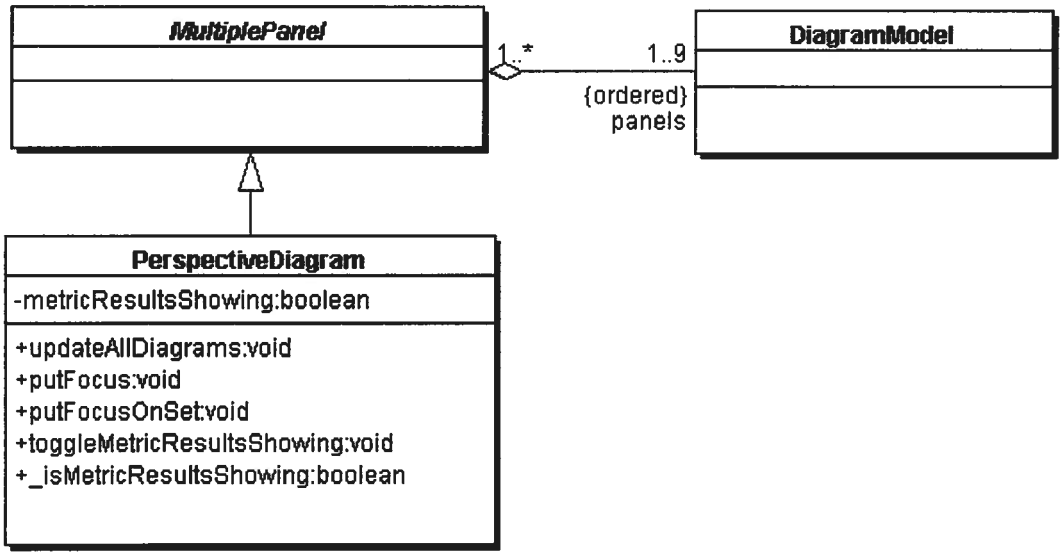


Figure 24: La classe *PerspectiveDiagram*

MultiplePanel. C'est la classe *MultiplePanel* qui réalise les fonctionnalités graphiques de la perspective, c'est-à-dire une fenêtre qui peut contenir jusqu'à 9 diagrammes. Elle maintient un ensemble ordonné de références à des éléments graphiques, et permet de disposer ces éléments dans une même fenêtre ayant 9 cases redimensionnables à l'aide d'une série de panneaux divisés (*javax.swing.JSplitPane*).

PerspectiveDiagram. Alors que la classe MultiplePanel s'occupe des fonctionnalités graphiques, la classe *PerspectiveDiagram* s'occupe des diagrammes qui composent la perspective. La classe contient un attribut *metricResultsShowing* qui définit si les résultats des métriques doivent être représentés ou non dans les diagrammes. Deux méthodes d'accès

+toggleMetricResultsShowing( ) : void et *+isMetricResultsShowing() : boolean*

sont aussi fournies pour manipuler cet attribut. Lorsque la méthode

+updateAllDiagrams( )

est appelée, la méthode *contentSetChanged( )* est appelée sur chacun des diagrammes pour qu'ils soient mis à jour. Dépendamment de la valeur de l'attribut *metricResultsShowing*, les résultats des métriques sont montrés ou non dans les diagrammes.

7.3.2 Synchronisation de la focalisation

Dans une perspective, les mêmes *ModelElement*s seront souvent représentés dans plusieurs diagrammes ou même dans tous les diagrammes. Un mécanisme permettant de retrouver rapidement un *ModelElement* dans chacun des diagrammes serait très utile pour exploré les résultats des différentes métriques pour ce *ModelElement*. C'est pourquoi MDP fourni une méthode qui permet de mettre en évidence un ou plusieurs *ModelElement*s dans tous les diagrammes simultanément. Les méthode

+putFocus(aModelElement : ModelElement)

pour un seul *ModelElement* et

+putFocusOnSet(aModelElementSet: Set)

pour un ensemble de *ModelElement*s appellent la méthode *triggerFocus* sur chaque diagramme. La méthode *triggerFocus* est une méthode abstraite de la classe *DiagramModel* (voir Figure 21) qui est implantée par chaque adaptateur.

7.3.3 Persistance des perspectives

Nous croyons que la création d'une perspective efficace se fera généralement après un long processus d'essais et d'erreurs, de comparaisons et de tests. C'est pourquoi il est important que les perspectives créés lors d'une session de travail soient facilement récupérables à la session suivante. Dans cet esprit, MDP permet de stocker les perspectives à même le dépôt de données. La Figure 25 montre comme cela est rendu possible.

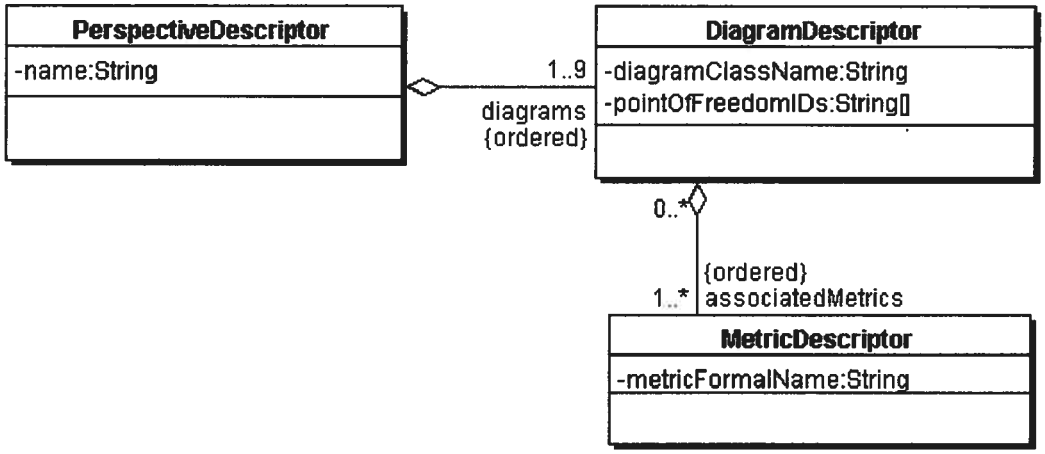


Figure 25: Les classes qui rendent possible la persistance des perspectives

Il aurait été possible de stocker directement les objets de type *PerspectiveDiagram*, *DiagramModel* et *Metric* mais puisque ces classes font partie d'un réseau d'objets complexes, énormément d'attributs auraient été stockés inutilement. Au lieu de cela, MDP fournit une structure de classes parallèle qui ne contient que le minimum d'information nécessaire pour rendre les perspectives persistantes.

Tout d'abord, la classe *MetricDescriptor* ne contient qu'un seul attribut de type String, *metricFormalName* qui contient le nom formel de la métrique tel que spécifié dans le fichier de définition de métriques.

La classe *DiagramDescriptor* ne contient que le minimum d'information permettant de recréer un diagramme avec les mêmes métriques associées aux mêmes points de liberté. Tout d'abord, l'attribut *diagramClassName* contient le nom complet de la classe java (*fully qualified name*) qui sert d'adaptateur au diagramme concret. La classe contient également un tableau de références vers des objets *MetricDescriptor* appelés *associatedMetrics* et un autre tableau de *String* appelé *pointOfFreedomIds* qui contient les identificateurs des points de liberté auxquels les métriques correspondantes aux objets MetricDescriptor sont associées. Donc, lorsqu'un objet *DiagramModel* est recréé à partir d'un objet *DiagramDescriptor* venant de la base de données, la métrique à la position *i* du tableau *associatedMetrics* se retrouvera associée au point de liberté dont l'identificateur *id* est la chaîne de caractères à la position *i* du tableau *pointOfFreedomIds*.

Finalement, dans la classe *PerspectiveDescriptor*, on retrouve une chaîne de caractères *name* qui contient le nom de la perspective ainsi qu'un tableau de longueur 9, *diagrams* qui contient des références vers des *DiagramDescriptor*. Puisque, dans une perspective, un diagramme peut occuper plusieurs cases, le tableau *diagrams* peut contenir plusieurs références au même objet *DiagramDescriptor*.

Chapitre 8: L'outil Thycho-metrics

Cette section vise à décrire Thycho-metrics, un outil basé sur MDP développé au cours de notre recherche. Cet outil sert de preuve de concept pour le cadre d'application MDP décrit au chapitre précédent. L'outil profite énormément des fonctionnalités de l'environnement SPOOL décrit au Chapitre 5.

8.1 Thycho-metrics, MDP et SPOOL

Des contraintes au niveau des ressources disponibles et du temps alloué pour compléter cette recherche ont empêché qu'une implémentation exacte du cadre d'application MDP tel que décrit dans les chapitres 6 et 7 soit complétée. Les deux sous-sections suivantes décrivent l'état réel de l'environnement SPOOL et du cadre d'application MDP ainsi que la description des fonctionnalités qu'ils fournissent à l'outil Thycho-metrics.

8.1.1 SPOOL

Comme discuté au Chapitre 5, SPOOL fournit un dépôt de données dont le schéma est basé sur les éléments du package *core* du méta-modèle UML version 1.1. Malheureusement, nos propos de la section 4.1 tendent à démontrer qu'un méta-modèle basé sur UML n'est pas idéal pour réaliser le calcul de métriques défini selon le formalisme de la Définition 14, d'autant plus que le dépôt de SPOOL comporte plusieurs singularités qui rendent effectif le problème exprimé par la Figure 6 et la Figure 7. Mais ceci n'empêche pas le dépôt de SPOOL d'être très utile à l'outil Thycho-metrics comme source de données.

L'autre contribution majeure de SPOOL se situe au niveau de la visualisation. SPOOL fournit plusieurs mécanismes très utiles à MDP comme le mécanisme de synchronisation modèle-vue et la gestion des fenêtres.

8.1.2 MDP

Dans son état actuel, MDP est implanté tel que décrit dans les deux chapitres précédents. La seule exception majeure est au niveau du calcul des métriques avec OCL. Des contraintes de temps ont empêché l'implémentation de cette partie de MDP. Les métriques sont présentement écrites en langage Java.

8.2 Démarrage de l'outil

La première chose à faire juste après le lancement de l'outil Thycho-Metrics est de charger une base de données.

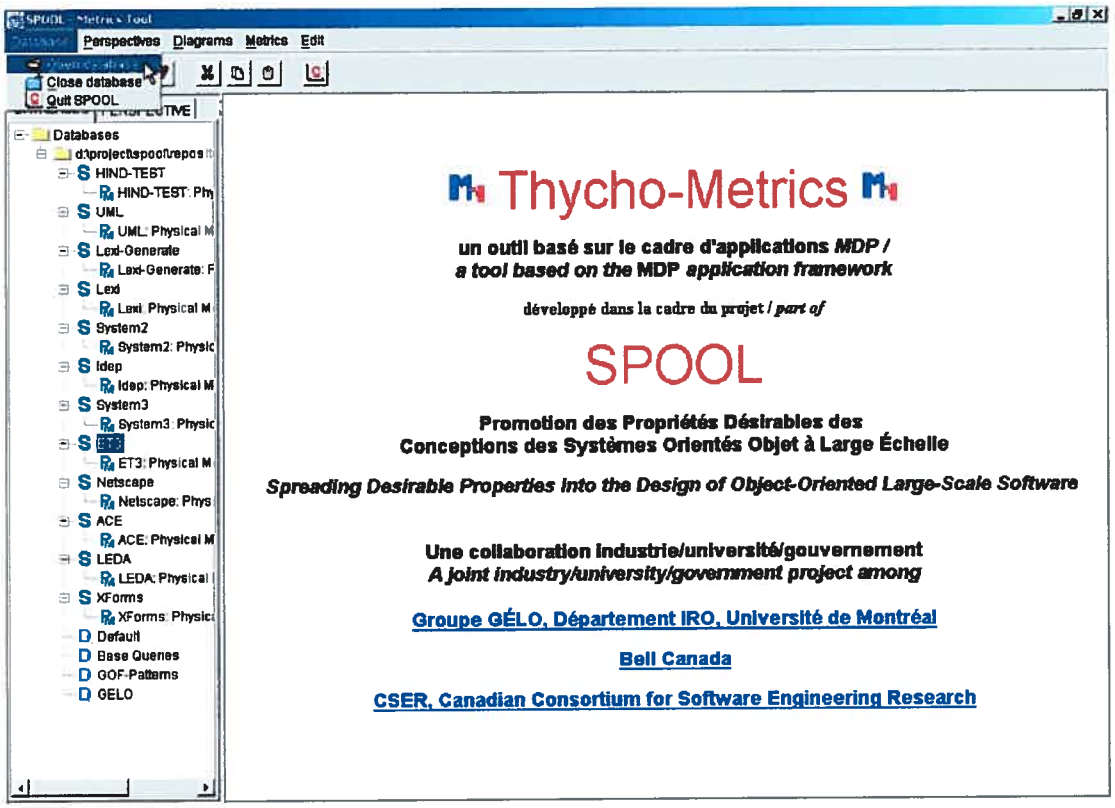


Figure 26: Écran de démarrage de l'outil Thycho-metrics

La Figure 26 montre l'allure de l'outil juste après le chargement de la base de données. Dans la partie droite de l'écran se trouve un fureteur montrant un fichier de

présentation de l'outil et du projet SPOOL. La partie gauche contient un panneau à onglets qui contient deux panneaux intitulés *DATABASES* et *PERSPECTIVES.* Celui qui est visible, *DATABASES* contient un arbre contenant tous les systèmes présents dans la base de données qui vient d'être chargée.

8.3 Auto-documentation des perspectives

Dans l'autre onglet de la fenêtre de la Figure 26, on retrouve, dans un arbre, la liste des perspectives qui ont été stockées dans la base de données lors d'utilisations précédentes (voir Figure 27). Pour l'affichage de cet arbre, seul des objets de type *MetricDescriptor*, *DiagramDescriptor*, *PerspectiveDescriptor* sont chargés depuis le dépôt.

Le fichier HTML qui se trouve à la droite de cette figure est généré automatiquement par MDP à la demande de l'utilisateur, en cliquant l'item *Show perspective information*. Ce fichier est généré à partir de plusieurs attributs dont l'attribut *description* des instances des classes *MetricDescriptor*, *DiagramDescriptor*, *PerspectiveDescriptor* impliquées. Sur la première ligne du fichier se trouve le nom de la perspective suivi de sa description. Ensuite les diagrammes sont énumérés en présentant tout d'abord le nom, la description, et un tableau énumérant toutes les métriques intégrées au diagramme. La première colonne donne le nom court de la métrique, la deuxième son nom long et ensuite le type de Tuple de la métrique, la description de la métrique, l'identificateur du point de liberté auquel la métrique est associée et la variable visuelle utilisée par le point de liberté. Ces informations permettent à l'utilisateur de choisir avec quelle perspective il veut travailler avant de la charger.

On peut également obtenir des informations sur un diagramme seul ou sur une métrique seule en cliquant sur l'item *Show diagram information* ou *Show metric information* dans le menu déroulant de l'arbre de la Figure 27.

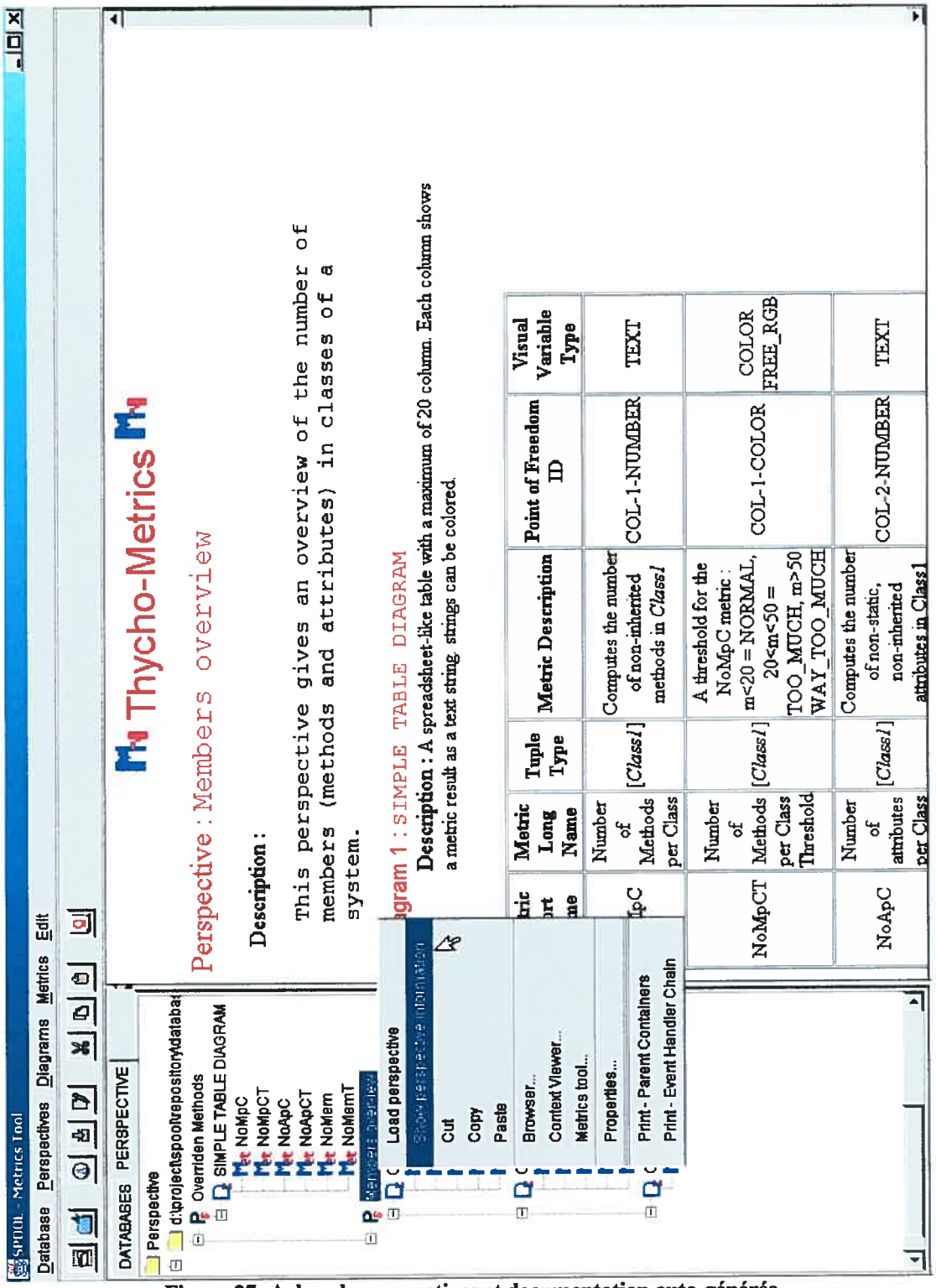


Figure 27: Arbre de perspectives et documentation auto-générée

Dans cette figure, on remarque une variable visuelle qui n'avait pas été mentionné auparavant soit *text*. C'est le cas dégénéré d'une variable visuelle qui exprime le résultat de la métrique littéralement, avec des lettres ou des chiffres, que ce soit un

nombre naturel, un nombre décimal ou une chaîne de caractères. Cette variable visuelle est bien sûr compatible avec tous les types d'échelle. Par exemple, les chiffres qui sont contenus dans la table sont des manifestations d'une variable visuelle *text*.

8.4 Chargement et remplissage d'une perspective

L'item *Load Perspective* du menu déroulant de la Figure 27 charge une perspective dans une nouvelle fenêtre comme celle de la Figure 30 mais vide, sans éléments et sans résultats de métriques visibles. Ici c'est la classe *PerspectiveDiagram* qui est instanciée à partir des informations contenues dans le *PerspectiveDescriptor*. Le *PerspectiveDescriptor* réfère à des *DiagramDescriptor*s qui eux réfèrent à des *MetricDescriptor*s. C'est dans cet ordre que les classes qui correspondent à ces descripteurs (*PerspectiveDiagram*, des sous-classes de *DiagramModel*, *Metric*) se font instancier par l'outil *Thycho-metrics* pour créer la perspective.

Initialement, les diagrammes de la perspective sont vides. En appuyant sur le premier bouton de la barre d'outils ou en cliquant l'item *Initialize* du menu *Perspective* (Figure 30), la fenêtre de dialogue montrée à la Figure 28 apparaît. Cette fenêtre permet de spécifier les *ModelElements* que l'on désire visualiser dans chaque diagramme. Pour chacun des diagrammes, les *contentTypes* (attribut de la classe *DiagramModel*, voir Figure 21) sont listés. MDP associe à chacun de ces *contentTypes* un ensemble qui contient des *ModelElements* ayant le type du *contentType*. On remplit chacun de ces ensembles en appuyant sur les boutons de la fenêtre de la Figure 28. Le *Design Browser* [70][71][27] apparaît alors sous la forme d'une nouvelle fenêtre de dialogue. Le *Design Browser* est un outil faisant partie de l'environnement SPOOL qui permet entre autres de naviguer à travers un système au moyen de plusieurs requêtes pré-définis. L'utilisateur doit effectuer les requêtes nécessaires pour obtenir que les *ModelElements* désirés soient affichées dans la partie droite du *Design Browser*. Seulement des *ModelElements* de type *contentType* doivent être choisis. Après, l'utilisateur appuie sur *OK*, le *Design Browser* disparaît et le nombre de *ModelElements* qui ont été choisis est affiché à coté du *contentTypes* correspondant. Lorsque les contenus de tous les ensembles ont été choisis,

l'utilisateur appuie sur *OK* et la fenêtre de remplissage disparaît. Le tableau *contentSets* de la classe *DiagramModel* (Figure 21) est maintenant initialisé.

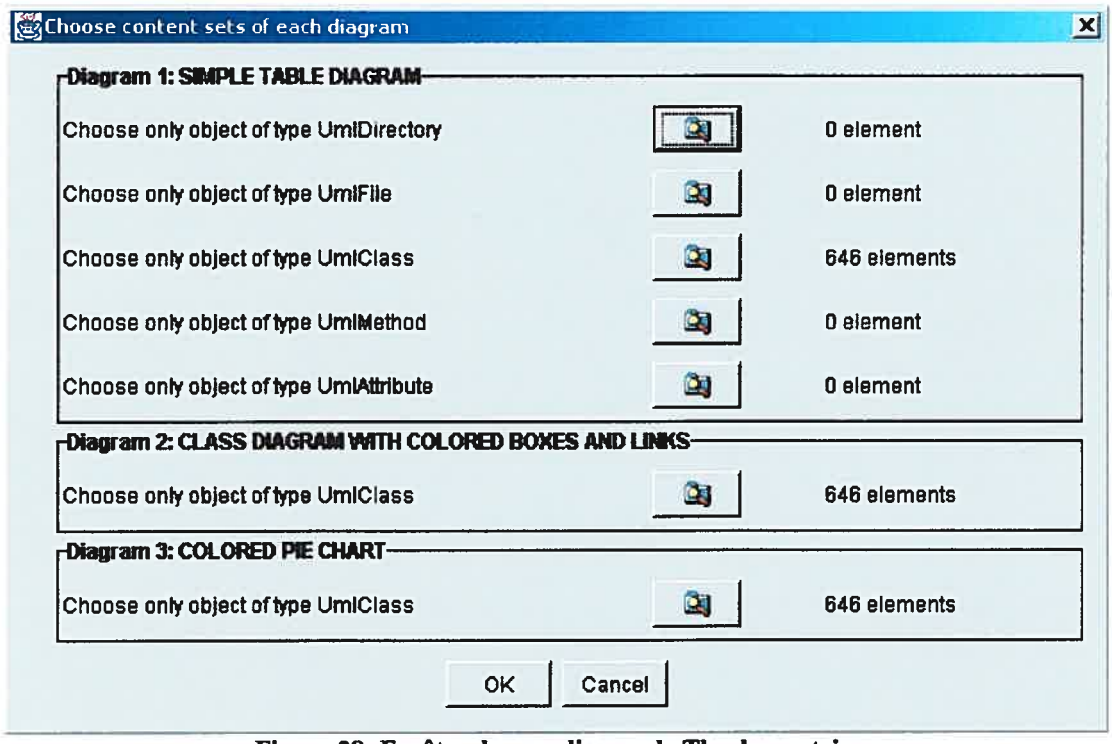


Figure 28: Fenêtre de remplissage de Thycho-metrics

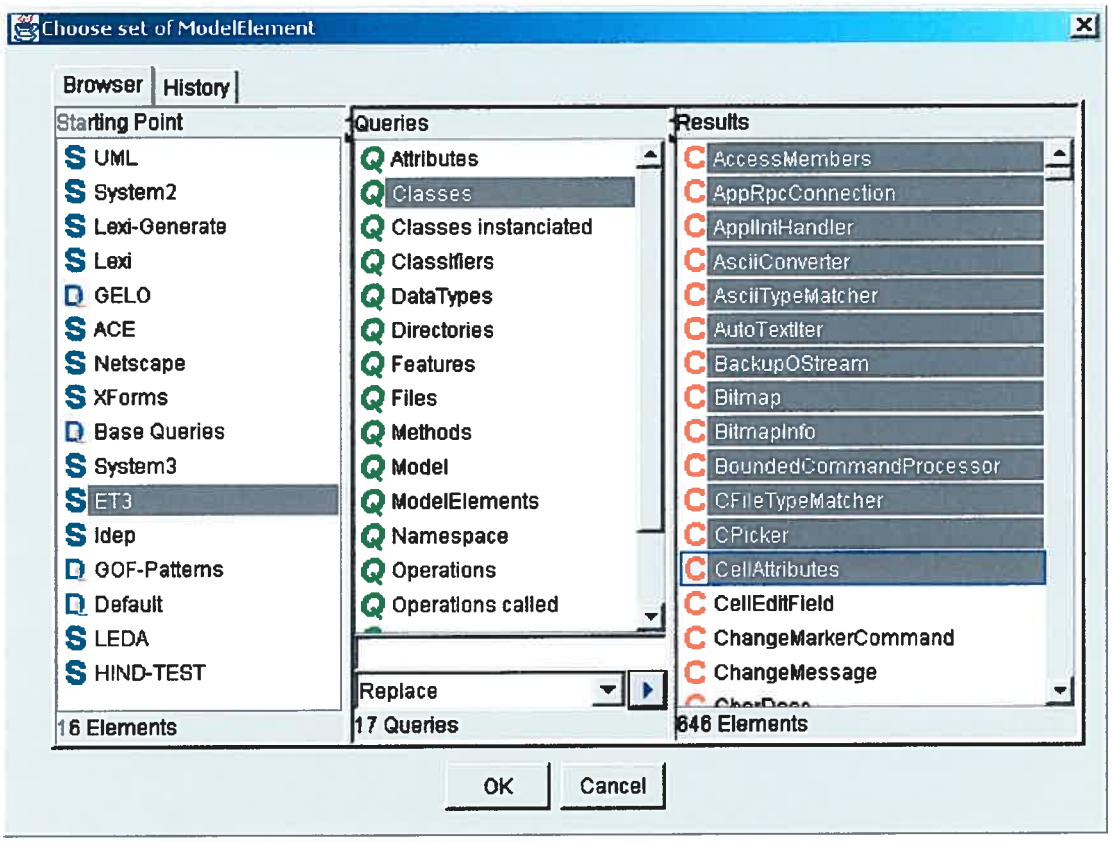
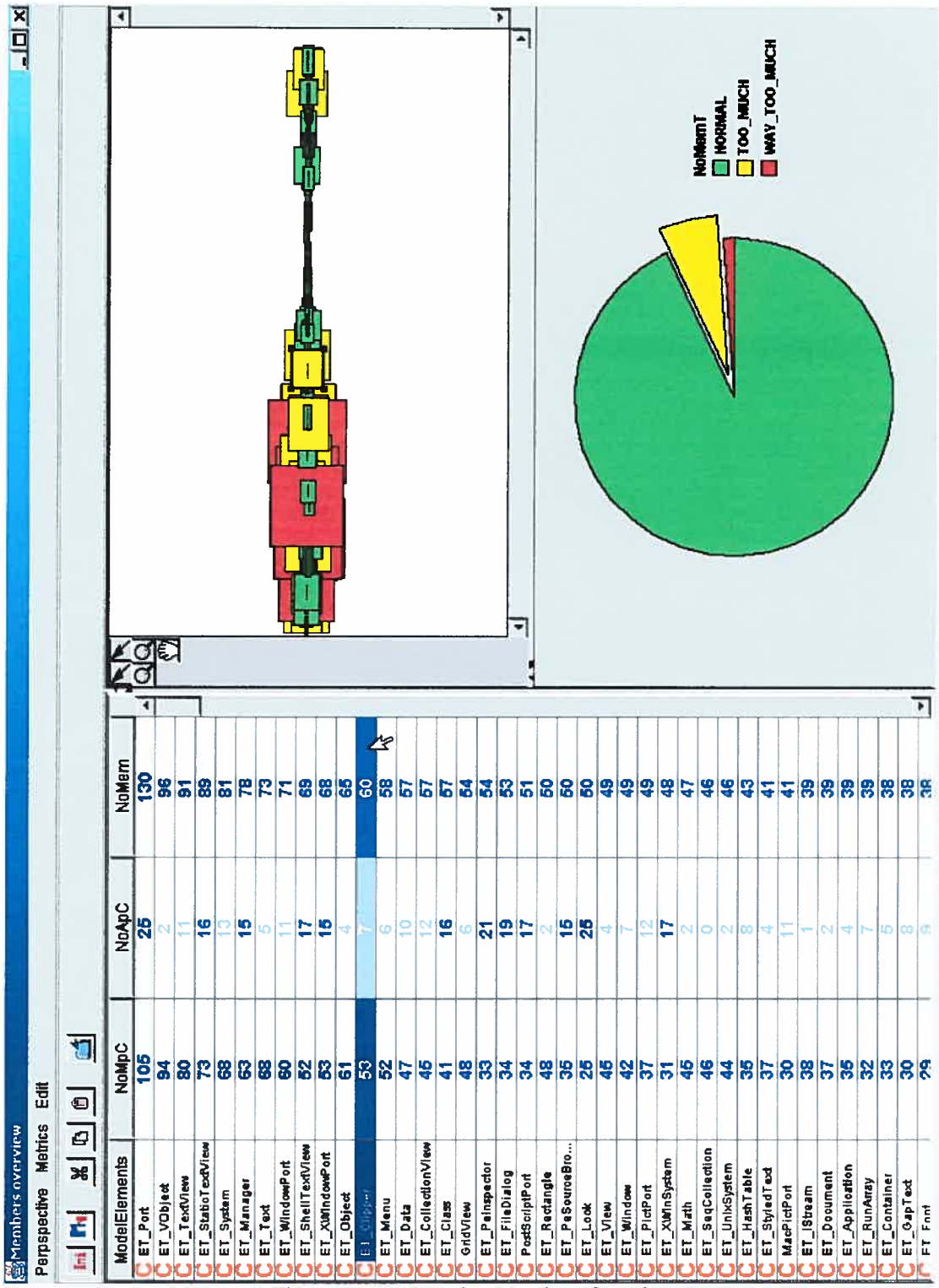


Figure 29: Le *Design Browser*

Thycho-metrics appelle alors la méthode *updateAllDiagrams* de la classe *DiagramModel*. Les *ModelElements* choisis sont alors affichés comme dans la Figure 30 sauf qu'initialement les résultats de métriques n'apparaissent pas. La table est vide, il n'y a pas de boîtes colorées autour des classes du diagramme de classe et la tarte est absente. Cela permet à l'utilisateur de MDP de consulter les diagrammes qui fournissent de l'information pertinente autre que les résultats de métriques comme, par exemple, le diagramme de classes sans la charge additionnelle des résultats de métriques. L'utilisateur doit cliquer sur le deuxième bouton de la barre d'outils pour obtenir ces résultats. Cet action appelle les méthodes *toggleMetricsResultShowing* et *updateAllDiagrams* de la classe *PerspectiveDiagram*.



ModelElements	NoMpC	NoApC	NoMem
ET_Port	105	25	130
ET_VObject	94	2	96
ET_TextView	80	11	91
ET_StaticTextView	73	16	89
ET_System	68	13	81
ET_Manager	63	15	78
ET_Text	68	5	73
ET_WindowPort	60	11	71
ET_ShellTextView	52	17	69
ET_XWindowPort	53	15	68
ET_Object	61	4	65
[illegible]	53	7	60
ET_Menu	52	6	58
ET_Data	47	10	57
ET_CollectionView	46	12	57
ET_Class	41	16	57
GridView	48	6	54
ET_Painspector	33	21	54
ET_FileDialog	34	19	53
PostScriptPort	34	17	51
ET_Rectangle	48	2	50
ET_PaSourceBro...	35	15	50
ET_Look	25	25	50
ET_View	45	4	49
ET_Window	42	7	49
ET_PictPort	37	12	49
ET_XWinSystem	31	17	48
ET_Math	46	2	47
ET_SeqCollection	46	0	46
ET_UnixSystem	44	2	46
ET_HashTable	35	8	43
ET_StyledText	37	4	41
MacPictPort	30	11	41
ET_IStream	38	1	39
ET_Document	37	2	39
ET_Application	35	4	39
ET_RunArray	32	7	39
ET_Container	33	5	38
ET_GapText	30	8	38
ET_Font	29	9	38

Figure 30: Perspective *Members Overview*

8.5 La perspective *Members overview*

La Figure 30 illustre la perspective *Members Overview* qui démontre la puissance de l'approche de MDP en donnant un aperçu rapide du nombre de membres (attributs et méthodes) dans le système ET3 [38]. Cette perspective contient les métriques énumérées dans le Tableau 16.

Nom court	Nom long	Type de Tuple	Type d'échelle
NoMem	Number of Members	[CLASS]	Absolue
NoMemT	Number of Members Threshold	[CLASS]	Ordinal
NoMpC	Number of Method per Class	[CLASS]	Absolue
NoMpCT	Number of Method per Class Threshold	[CLASS]	Ordinal
NoApC	Number of Attribute per Class	[CLASS]	Absolue
NoApCT	Number of Attribute per Class Threshold	[CLASS]	Ordinal
NIM	Number of Inherited Method	[CLASS,CLASS]	Absolue
NIMT	Number of Inherited Method Threshold	[CLASS,CLASS]	Ordinal

Tableau 16 : Métriques de la perspective Members Overview

Elle contient également trois des diagrammes qui ont été réalisés pour l'outil Thychometrics. Dans ces diagrammes les mêmes métriques sont visualisées de façon redondante et de différentes façons. La métrique NoMemT, par exemple, se retrouve dans les trois diagrammes. Les trois sous-sections suivantes décrivent les trois diagrammes que l'on retrouve dans la perspective *Members Overview.*

SIMPLE TABLE DIAGRAM

Le premier diagramme, appelé *SIMPLE TABLE DIAGRAM*, est une table du genre feuille de calcul qui peut compter jusqu'à 20 colonnes. Chaque colonne fournit deux points de liberté. Le premier utilise une variable visuelle de type *text*. C'est la chaîne de caractères qu'il y a dans chaque case. Le deuxième point de liberté utilise la variable visuelle *couleur*. C'est la couleur de la chaîne de caractères de chaque case. Comme mentionné précédemment, la variable visuelle *text* supporte tous les types d'échelles car tous les types de valeurs peuvent prendre une forme textuelle. Il n'y a donc pas de restrictions sur le genre de métrique que le premier point de liberté de chaque case peut supporter. Pour le deuxième, le tableau 10 nous dit que la variable

visuelle *couleur* est mieux appropriée pour les échelles de type *Nominal* et *Ordinal* mais pas trop mauvaise pour les autres. Les identificateurs des premiers points de liberté de chaque colonne est COL-*n*-NUMBER, et les identificateurs des deuxièmes points de liberté de chaque colonne est COL-*n*-COLOR où *n* est un nombre de 1 à 20. Donc, on arrive à 40 points de liberté en tout pour ce diagramme.

Au niveau des types de tuple, ce diagramme ne permet d'afficher qu'un seul *ModelElements* par rangée; ce sont tous des points de liberté du premier degré. Pour des raisons pratiques les types de tuples supportés ont été restreints à **[CLASS]**, **[DIRECTORY]**, **[FILE]**, **[METHOD]**, **[ATTRIBUTE]** dans la Figure 28 mais tout type de tuple à un élément peut être supporté. Les métriques intégrées doivent donc avoir un type de tuple compatible avec l'un de ces cinq types de tuple. C'est le cas pour toutes les métriques de ce diagramme, NoMem, NoMemT, NoMpC, NoMpCT, NoApC, NoApCT, qui ont toutes comme type de tuple **[CLASS]**.

Ce diagramme permet également quelques interactions. Cliquer sur une entête permet de trier la liste, en ordre alphabétique (ou l'inverse) pour la colonne des *ModelElements* ou en ordre de valeurs (ou l'inverse) pour les autres colonnes.

Concrètement, dans le diagramme, seulement six points de liberté sont utilisés : NoMpC et NoMpCT dans la première colonne, NoApC et NoApCT dans la deuxième colonne, NoMem et MoMemT dans la troisième colonne. Pour les trois métriques de type *Threshold*, le diagramme utilise les couleurs fournies par MDP en utilisant la méthode *getFormatedValue* de la classe *DiagramModel*.

CLASS DIAGRAM WITH COLORED BOXES AND LINKS

Le deuxième diagramme CLASS DIAGRAM WITH COLORED BOXES AND LINKS procure un diagramme de classes avec cinq points de liberté soit la largeur et la hauteur des boîtes enrobant les classes, la couleur de ces boîtes ainsi que la largeur du lien d'héritage qui joint les classes et sa couleur. Donc deux points de liberté utilisent la variable visuelle *couleur* et les trois autres la variable visuelle *taille*. Les trois premiers points de liberté ont comme type de tuple **[CLASS]** et les deux autres, la largeur et la couleur des liens, ont comme type de tuple **[CLASS, CLASS]**.

Puisque le but de cette perspective est en premier lieu d'explorer le nombre de membres dans chaque classe, la métrique NoMem a été intégré à la hauteur *et* à la largueur des boîtes autour des classes, tandis que la métrique NoMemT est visualisé avec la couleur de ces boîtes. Les points de liberté au niveau de la largeur et de la couleur des liens sont les seuls où l'on peut faire la visualisation des métriques NIM et NIMT car ce sont les séuls à être visuellement associés à deux classes, i.e. à avoir un type de tuple **[CLASS,CLASS]**.

Ce diagramme présente également un modèle d'interactions complexe. Dans la Figure 30, l'arbre d'héritage montre chacune des 646 classes du système ET3. L'utilisateur peut faire un zoom sur la région de l'arbre qu'il désire et naviguer dans le diagramme à l'aide des barres de défilement. Il peut également manipuler la position des nœuds de l'arbre à sa guise.

COLORED PIE CHART

Le troisième diagramme de la perspective *Members Overview* est un graphique en tarte. Ce diagramme constitue un bel exemple de la flexibilité que procure MDP. Il possède un seul point de liberté qui utilise une variable visuelle de type *couleur* et dont le type de tuple est **[PACKAGE]**, un package étant un ensemble de *ModelElements* (voir Chapitre 5). Pour une métrique dont le type d'échelle est nominal ou ordinal, le diagramme montre quelle proportion des *ModelElements* ont leur résultat dans chacune des catégories.

Le diagramme fournit également une légende qui montre la chaîne de caractères qui correspond à chaque valeur dans l'échelle.

Synchronisation de la focalisation

La Figure 30 démontre également le mécanisme de synchronisation de la focalisation. Comme mentionné dans la section 7.3.2 chaque diagramme doit être capable de mettre en évidence des *ModelElements* à la demande MDP (voir Figure 21). Les trois diagrammes s'acquittent de cette tâche de trois façons bien différentes. La table sélectionne les *ModelElements* et place la barre de défilement de façon à ce que le premier *ModelElement* sélectionné de la liste soit au haut de la page, le diagramme de

classes enrobe les boîtes des classes avec un carré noir et le graphique en tarte sépare les sections dans lesquels se trouvent les *ModelElements* à mettre en évidence.

8.6 Création d'une nouvelle perspective

Thycho-metrics profite pleinement des capacités de MDP pour permettre la création de nouvelles perspectives et pour permettre la modification de perspectives existantes. La Figure 31 montre l'éditeur de perspectives de *Thycho-metrics* avec la perspective *Members Overview* décrite dans la section précédente chargée.

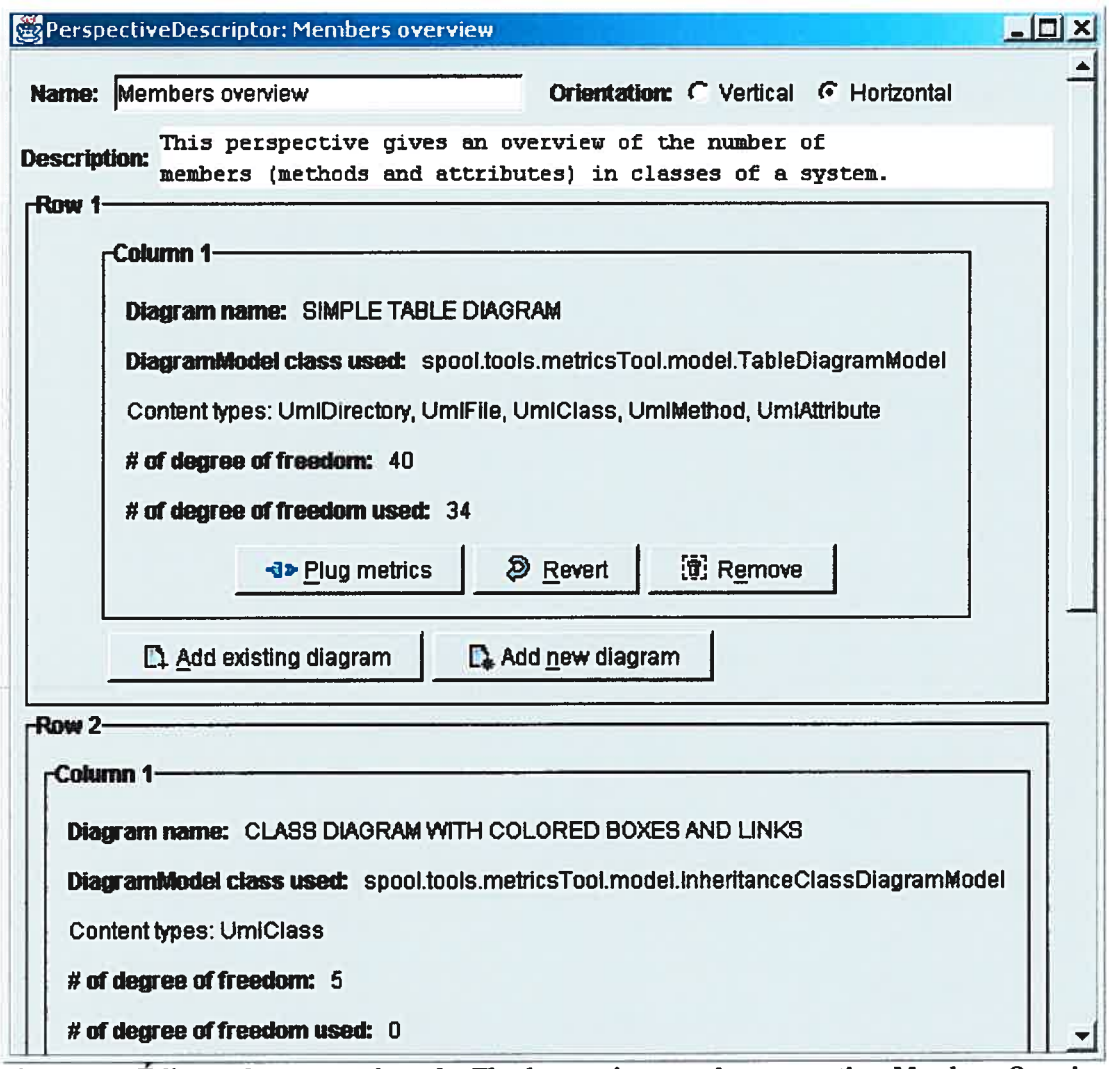


Figure 31: Éditeur de perspectives de *Thycho-metrics* avec la perspective *Members Overview* chargée.

Les informations présentées dans cette fenêtre sont contenues dans le descripteur de perspectives, les descripteurs de diagrammes associés et dans les classes adaptateur

utilisées par ces diagrammes. Le nom et la description de la perspective peuvent être modifiés directement en modifiant les champs *Name* et *Description*. Ensuite, l'éditeur de perspectives sépare la perspective en trois rangées (*row*) qui peuvent chacune contenir jusqu'à trois diagrammes, soit un par colonne (*column*).

Pour chacun des diagrammes, les informations présentées sont celles contenues dans le descripteur de diagramme ainsi que dans la classe adaptateur utilisée. Premièrement, le nom du diagramme en question ainsi que la classe Java utilisée comme adaptateur sont données. Ensuite, les types de *ModelElements* qui peuvent être représentés dans le diagramme sont donnés (*Content Types*). Finalement, le nombre total de points de liberté ainsi que le nombre de points de liberté utilisés sont donnés.

Pour modifier les métriques associées au différents points de liberté d'un certain diagramme, l'utilisateur doit cliquer sur le bouton *Plug Metric* qui se trouve au bas de chaque description de diagramme. Le bouton *Revert* peut être utilisé pour restaurer l'état du diagramme à l'état où il était au moment du lancement de la fenêtre, alors que le bouton *Remove* enlève le diagramme de la perspective.

Pour ajouter un nouveau diagramme à la perspective, il suffit de cliquer sur le bouton *Add new diagram* au bas de la section qui correspond à la rangée dans laquelle on désire ajouter ce diagramme. La Figure 32 montre la fenêtre de dialogue qui permet d'ajouter un nouveau diagramme à une perspective. Cette figure est présentée dans le contexte de la création du diagramme *CLASS DIAGRAM WITH COLORED BOXES AND LINKS*.

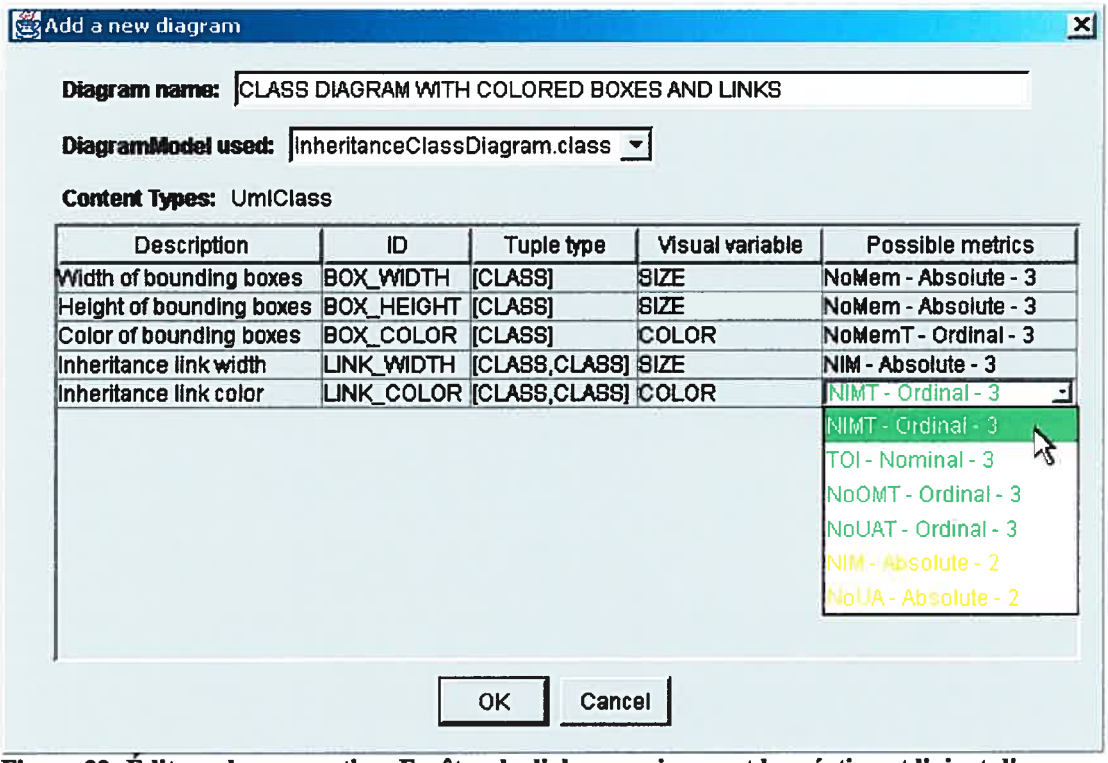


Figure 32: Éditeur de perspective: Fenêtre de dialogue qui permet la création et l'ajout d'un nouveau diagramme à la perspective.

La première chose à faire pour créer un nouveau diagramme est de lui donner un nom, nom qu'on inscrit dans le champ *Name*. Ensuite il faut choisir la classe adaptateur utilisée pour la communication avec la composante qui implémente concrètement le diagramme. Dans le menu déroulant se trouve la liste de toutes les sous-classes de *DiagramModel* trouvée dans le *CLASSPATH*.

Un fois l'adaptateur choisi, le champ *Content Types* et la table des points de liberté sont initialisés automatiquement d'après les informations contenus dans l'adaptateur. L'utilisateur n'a plus qu'à choisir quelle métrique il veut associer à chaque point de liberté en les choisissant dans les menus déroulants. Dans ces menus déroulants, les métriques sont classées en ordre d'adéquation pour le point de liberté. Les métriques qui ne peuvent pas être associées au point de liberté soit parce que les types de tuple sont incompatibles soit parce que l'association est impossible ne sont pas affichées dans le menu. Les métriques ayant un indice de 3 selon la logique du Tableau 10 sont affichées dans le haut de la liste en vert, les métriques ayant un indice de 2 sont affichées ensuite en jaune et, celles qui ont un indice de 1 sont affichées en rouge dans le bas de la liste.

Une fois un diagramme créé dans l'éditeur de perspectives il peut être réutilisé dans d'autres perspectives. Il suffit de cliquer sur le bouton *Add existing diagram* (voir Figure 31) et choisir le bon diagramme dans la liste. Cette liste est constituée à partir de tous les objets *DiagramDescriptor* se trouvant dans la base de données.

Chapitre 9 : Discussion

Dans ce chapitre, nous discutons plus en profondeur certains aspects de nos travaux. La section 9.1 présente une synthèse qui met l'emphase sur les contributions réelles apportées par ceux-ci. La section 9.2 examine plus en détails certains points précis de l'approche que nous avons proposée dans ce mémoire. Finalement, la section 9.3 discute des validations que nous avons effectuées.

9.1 Synthèse

9.1.1 Analyse des problèmes chroniques des métriques

Le domaine des métriques souffre de plusieurs problèmes chroniques. Nous caractérisons ces problèmes de *chroniques* car la grande majorité des chercheurs et professionnels les ont acceptés et ne tentent plus de les régler, s'ils ont jamais essayé. Ainsi la plupart des travaux qui présentent des résultats de métriques ne peuvent être reproduits à l'extérieur de l'environnement technique où ils ont été faits. Sans cette possibilité de reprendre ou de vérifier les résultats, la recherche dans le domaine des métriques se retrouve dans un cul-de-sac perpétuel pendant qu'au niveau de l'industrie le scepticisme fait tranquillement place à l'indifférence et même au cynisme.

Nous croyons que cet état des choses n'est pas dû à la mauvaise volonté mais bien à deux facteurs bien distincts. Le premier, nous croyons, est d'ordre technique. Comme expliqué dans la section 4.1, la pluralité des langages de programmation, des environnements de développement et des environnements de maintenance peut sembler être un obstacle insurmontable à la standardisation du processus de calcul de métriques. Le deuxième facteur serait d'ordre idéologique. En informatique, l'une des

méthodes les plus utilisées pour gérer la complexité est l'abstraction, une méthode qui consiste à cacher les détails pour mieux mettre en évidence les choses importantes. Dans cet esprit, il semble naturel de définir les métriques avec de vagues concepts orientés objet et de prétendre qu'elles s'appliquent également bien à tous les langages de programmation puisque ces langages ne diffèrent que dans les détails. Bien sûr, cette façon de définir les métriques va directement à l'encontre de la théorie de la mesure qui, elle, définit le concept de mesure dans un cadre mathématique strict.

L'une des contributions majeures de ce mémoire est de présenter une approche *honnête* par rapport aux métriques, honnête parce qu'elle reconnaît leurs limites inhérentes en tant que mesure et parce qu'elle propose une solution qui en tient compte. Bien que nous ayons ici affaire qu'à des problèmes collatéraux qui débordent quelque peu du sujet central de ce mémoire qui est la visualisation des métriques orientées objet, nous croyons qu'il n'était ni possible, ni désirable d'en faire abstraction une fois de plus.

Ainsi, l'approche face aux métriques présentée dans ce mémoire au moyen de la Définition 14 et des définitions sur lesquelles elle dépend ne doit pas être vue seulement comme une solution au problème de la visualisation des métriques mais bien comme une solution au problème de la *définition* des métriques, problème qui, selon nous, est responsable du faible degré d'acceptation que la discipline des métriques a reçu historiquement.

Concrètement, ce mémoire propose les solutions suivantes :

- Définir les métriques comme étant des *mesures* au sens de la théorie de la mesure. Pour une métrique donnée l'ensemble d'objet empiriques est un ensemble de *tuples* de *ModelElements* et l'ensemble d'objets mathématiques dépend du type de l'échelle de la métrique (voir Tableau 9).

- Définir les métriques de façon formelle dans un document XML (voir section 6.1.5)

- Utiliser le langage OCL pour définir la fonction qui calcule les métriques ainsi que les conditions d'applicabilité de la métrique.

- Reconnaître que les métriques ne peuvent être définies que pour un méta-modèle, un seul langage de programmation et une seule projection bien précise entre le langage et le méta-modèle.

Nous croyons que respecter ces propositions pourrait régler les problèmes mentionnés ci-dessus.

9.1.2 MDP

La plus grande contribution de ce mémoire est le cadre d'application MDP décrit dans les chapitres 6 et 7. MDP trouve sa nécessité dans plusieurs réalités bien établies du monde des logiciels. Premièrement, si on veut contrôler les logiciels il faut parvenir à les mesurer[34]. Les logiciels étant de plus en plus grand, la quantité de données générée par ces mesures devient énorme. D'un autre coté, la visualisation d'information est reconnue comme une technique efficace pour analyser simultanément une grande quantité de données. Tout ceci implique qu'il faut visualiser les résultats des métriques.

Maintenant, il existe un grand nombre de techniques de visualisation qui, de façon évidente, ne sont pas toutes également efficaces pour les différentes métriques existantes. Trouver la visualisation la plus efficace pour une ou plusieurs métriques données se fera le plus souvent par un processus d'essais et d'erreurs.

C'est ici que MDP intervient. La stratégie employée par MDP pour aider à la création de visualisations efficaces consiste à séparer les métriques et les techniques de visualisation en différentes classes pour ensuite étudier les affinités entre les classes des deux groupes.

Du coté des métriques on utilise la théorie de la mesure pour faire la classification. Les métriques sont classées selon le type de leur échelle.

Du coté de la visualisation, MDP introduit le concept de *point de liberté*. Un point de liberté est une caractéristique d'un diagramme qui peut varier selon une valeur externe. On appelle ces caractéristiques *variables visuelles*. La classification des points de liberté se fait de part la variable visuelle qu'ils utilisent (couleur, taille, position, etc.).

En plus de cela, les métriques et les points de liberté sont tous deux classés de part le type de leur tuple de ModelElement. Une métrique dépend d'un tuple de ModelElement car une métrique est une mesure sur un nombre fini de ModelElements qui ont chacun un type précis . Un point de liberté dépend d'un tuple de ModelElement car sa variable visuelle est visuellement associée à un nombre fini de ModelElements. MDP fournit une logique stricte qui détermine quel mariage métrique/point de liberté est *possible* par rapport à leur type de tuple (voir section 6.2.3).

L'affinité entre les classes de métriques et de points de liberté a été étudiée de façon subjective et un *indice d'adéquation* entre 0 et 3 a été attribué à chaque couple (Tableau 10). Ces données constituent une logique floue qui donne un niveau de préférence à chaque mariage métrique/point de liberté. Elles sont intégrées à MDP et peuvent être utilisées librement par les outils basé sur MDP.

MDP suggère également l'utilisation de concepts psychophysiques pour améliorer l'efficacité des visualisations (voir sections 2.2.1, 7.2.3 et 7.2.4). Bien que ces concepts ne soient pas utilisés intensément dans ce mémoire, leur utilité est clairement démontrée.

Finalement, MDP propose un mécanisme d'intégration des diagrammes appelé perspective (voir section 6.3). Ce mécanisme permet à l'utilisateur d'un outil basé sur MDP de présenter plusieurs diagrammes complémentaires dans une même fenêtre, tout en permettant certaines interactions entre ces diagrammes.

9.1.3 Thycho-metrics

L'outil Thycho-metrics constitue une validation de l'approche proposée dans ce mémoire. Le bon fonctionnement de cet outil démontre que dans MDP, les bonnes abstractions ont été faites pour séparer les fonctionnalités fournies par MDP des choses concrètes comme les diagrammes, les métriques et les interfaces graphiques.

Le Chapitre 8 montre comment, concrètement, les fonctionnalités fournies par MDP peuvent être utilisées par un outil. Thycho-metrics fait usage de toutes les principales fonctionnalités fournies par MDP :

- calcul de métriques
- persistance des résultats
- table des indices d'adéquation
- données optimales pour la variable visuelle *couleur*
- données optimales pour la variable visuelle taille
- création, édition et stockage des perspectives et des diagrammes
- mécanisme de perspective
- synchronisation de la focalisation entre les diagrammes d'une perspective

9.2 Discussion

9.2.1 Méta-modèles

Choix d'un méta-modèle

Dans cet ouvrage, avec de bonnes raisons à l'appui, nous suggérons que les métriques soient explicitement définis en fonction d'un méta-modèle en particulier.

Nous proposons également que ces méta-modèles soient des instances du MOF. Loin d'être gratuite, cette restriction nous permet d'imposer l'utilisation d'OCL comme langage de définition des métriques. Puisque OCL est compatible avec n'importe quel

méta-modèle qui est une instance de MOF, on s'assure ainsi que toutes les métriques puissent être définies avec un seul langage.

Cet ouvrage ne propose cependant pas un méta-modèle en particulier comme solution universelle. Étant donné sa grande popularité, le méta-modèle UML semble être un choix évident. Cependant, en accord avec plusieurs auteurs [23], nous croyons que l'utilisation méta-modèle UML n'est peut-être pas la bonne solution pour les tâches de maintenance comme le calcul de métriques. Mais tout de même, si on s'en tient à la Définition 13 et à la Définition 14, il suffit de définir, de façon formelle, une projection entre le méta-modèle UML et un langage de programmation pour rendre l'utilisation du méta-modèle UML convenable selon les standards fixés par MDP. Ici il est important de comprendre que l'on n'obtiendrait pas un méta-modèle universel pour tous les langages de programmation car l'adaptation de UML pour un langage de programmation nécessite souvent l'utilisation de ses mécanismes d'extension ce qui crée un nouveau méta-modèle distinct des autres méta-modèles adaptés pour d'autres langages.

Pour ce qui en est du calcul des métriques, nous croyons que la meilleure solution est celle qui réduit au minimum l'écart sémantique entre le vocabulaire utilisé pour définir la métrique (noms des objets et des attributs dans la requête OCL qui définit la métrique) et les noms des éléments du langage de programmation. On accomplit cela en définissant un méta-modèle dont la sémantique des éléments est exactement celle des éléments du langage de programmation pour lequel on a défini des métriques (voir section 6.1.4 sous-section *Choix du méta-modèle et spécification de la projection*).

Niveau de détail

Un autre facteur peut venir brouiller les cartes, celui du niveau de détail qu'on s'attend du méta-modèle. Les deux principales catégories de métrique de produit sont les métriques de design et les métriques de code source. Les concepteurs d'un outil conçu seulement pour les métriques de design pourraient vouloir éviter d'utiliser un méta-modèle trop lourd en faisant abstraction des éléments de très bas niveau qui ne

sont nécessaires qu'aux métriques de code source. D'un autre coté, on voudrait bien sûr que les métriques de design définies pour un tel méta-modèle soient réutilisables dans un outil qui utilise la version complète du méta-modèle, ce qui viole les restrictions au niveau de la définition des métriques puisqu'elles se retrouvent définies pour deux méta-modèles différents.

Une caractéristique très utile pour un méta-modèle qui serait choisi pour un outil qui calcule les métriques serait qu'il puisse rester intègre et fonctionnel si on abandonne les éléments de bas niveau. De cette façon une même métrique pourrait être réutilisée avec des versions d'un même méta-modèle qui ne diffèrent que de part leur niveau de détail.

Nécessité des standards

Puisque les métriques et leurs résultats ne peuvent être repris que par ceux qui utilisent le même méta-modèle, un certain travail de standardisation dans le monde des outils à métriques s'impose. Le but de cette standardisation serait de restreindre au minimum le nombre de méta-modèles différents utilisés dans le but de maximiser la portée des métriques et de leurs résultats. Cela nécessitera bien sûr quelques ajustements techniques mais surtout beaucoup de bonne volonté.

9.2.2 Langage de définition des métriques

Mise à part son affinité avec tous les méta-modèles qui sont des instances de MOF, OCL a plusieurs qualités qui font que ce langage s'impose comme langage pour la définition des métriques :

- Les concepts de pré-condition et de post-condition sont dès le départ très appropriés pour modéliser les deux sortes d'expressions dont MDP a besoin soit les conditions d'applicabilités et les métriques elle-même (voir 6.1.4 sous-section *MOF et OCL, une solution presque complète*).

- Il existe déjà un important support logiciel pour ce langage sous la forme d'analyseur syntaxique et d'évaluateur d'expressions.

- C'est un langage déjà assez connu surtout à cause de sa relation avec le très populaire standard UML.

Pour ces trois raisons, OCL nous a semblé être de loin la meilleure solution.

9.2.3 Classification des métriques

Puisque l'un des objectifs de base de l'approche proposée avec MDP est de baser fortement le concept de métrique sur la théorie de la mesure, le choix des types d'échelle comme critère de classification était un choix évident.

Ce choix nous semble toujours être le bon. Le concept de type d'échelle semble réellement incorporer les facteurs qui déterminent le genre de variable visuelle approprié pour une certaine mesure.

9.2.4 Classification des points de liberté

C'est sous l'influence de plusieurs autres auteurs [7][87] que le concept de variable visuelle (couleur, taille, orientation, forme, texture, courbure) a été utilisé pour classifier les points de liberté. Loin d'être un succès retentissant, il apparaît maintenant que beaucoup plus de travail sera nécessaire avant que MDP puisse vraiment, un jour, permettre l'intégration de n'importe quel diagramme avec la simple création d'une courte classe adaptateur.

Un exploration rapide des composantes logicielles de visualisation offertes sur le marché est suffisante pour réaliser que la variable visuelle *couleur* est utilisée de beaucoup de façons différentes qui ne peuvent pas toutes être décrites par les moyens qu'on retrouve dans MDP présentement. Il nous apparaît maintenant que la plupart des variables visuelles doivent être divisées de nouveau en plusieurs sous-catégories et l'affinité des différents types d'échelle doit être réévaluée pour chacune de ces nouvelles sous-catégories.

La classification des points de liberté de par leur variable visuelle demande une étude à grande échelle de toutes les composantes de visualisation sur le marché. Seulement après qu'une telle étude ait été effectuée pourra-t-on définir correctement les

catégories et sous-catégories de variables visuelles appropriées pour MDP. Il sera sans doute également nécessaire de trouver un système de classification plus approprié que celui présentement en place dans MDP.

9.2.5 Principes psychophysiques

Pour améliorer l'efficacité des visualisations de résultats de métriques, ce mémoire suggère l'utilisation de concepts psychophysiques. En particulier, la psychophysique peut aider à choisir la manière dont les variables visuelles sont choisies. Par exemple, dans la section 7.2.4, une méthode pour choisir des valeurs pour la variable visuelle couleur dans le cas d'une association avec une échelle ordinale ou nominale est exposée. Dans MDP les fonctionnalités de ce genre sont intégrées dans la hiérarchie de classes *ResultFormatter* Figure 23.

Ce mémoire ne fait qu'effleurer le sujet de l'utilisation de la psychophysique pour optimiser la visualisation des résultats des métriques. Il nous semble cependant que cet axe de recherche soit très prometteur. Dans ce cas-ci, le défi consiste à débusquer des résultats d'expériences sur la psychophysique de la visualisation qui soient pertinents à la visualisation de métriques pour ensuite les adapter à MDP.

9.3 Validation

Comme mentionné dans la section précédente, beaucoup plus de recherche sera nécessaire avant que MDP en arrive à pouvoir vraiment intégrer n'importe quelle composante logicielle de visualisation. Les problèmes les plus évidents de MDP dans son état actuel se situent au niveau de la classification des variables visuelles. Nous croyons cependant qu'il est possible de catégoriser toutes les variables visuelles avec une étude appropriée.

Pour ces raisons, nous considérons que MDP tel que décrit dans ce mémoire est encore loin de l'étape de la validation. Cependant l'approche générale que propose MDP a été validée par la création de l'outil Thycho-metrics.

Chapitre 10 : Conclusion

L'initiative de créer un cadre d'application pour la visualisation des métriques orientés objet provient, à l'origine, de la volonté de généraliser les idées novatrices qu'on retrouve dans l'outil *Code Crawler* quant à la visualisation des métriques.

Les résultats obtenus jusqu'à présent sont encourageants. Nous sommes parvenus à réduire à un minimum l'effort nécessaire à l'intégration d'un nouveau diagramme à MDP ce qui était l'un de nos buts premiers. Nous avons également démontré que la logique d'intégration des métriques peut être facilement rendue disponible à plusieurs outils en réalisant cette logique sous la forme d'un cadre d'application.

Peut-être la tâche la plus urgente dans l'éventualité où la recherche sur MDP serait reprise, serait de terminer l'implémentation de MDP tel que présentement décrit dans les chapitres 6 et 7, la lacune majeure étant l'absence d'un évaluateur d'expression OCL pour permettre le calcul de métriques spécifiées en langage OCL comme le recommande MDP. Par la suite, il faudrait mener une vaste étude des composantes logicielles disponibles sur le marché et, avec les résultats, définir un nouveau Tableau 10 plus riche et plus précis. Par la suite, on pourrait réellement mettre MDP à l'épreuve en ajoutant à un outil basé dessus, Thycho-metrics par exemple, un grand nombre de visualisations très diverses. L'efficacité des visualisations pourrait alors être améliorée encore plus en intégrant des considérations psychophysiques à MDP.

Bibliographie

[1] Almare Diagrammer. Documentation disponible en ligne: <http://www.almarevisuals.com/en/diagrammer/index.html>, novembre 2002.

[2] Antoniol G, Fiutem R, Cristoforetti L. Using metrics to identify design patterns in object-oriented software. In *Proceedings of the Fifth International Software Metrics Symposium* (Metrics 1998), pages 23-34, Bethesda, MD, November 1998.

[3] Baroni AL, Braz S, Abreu FBe. Using OCL to formalize object-oriented design metrics definitions. In *Proceedings of the Sixth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Malaga, Spain, June 2002.

[4] Bansiya J, Davis CG. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1): 4-17, January 2002.

[5] Bassil S. *Évaluation qualitative et quantitative d'outils de visualisation logicielle.* Master's thesis, Université de Montréal, Montréal, Québec, Canada, décembre 2000.

[6] Bassil S, Keller RK. Software visualization tools: Survey and analysis. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'2001)*, pages 7-17, Toronto, ON, May 2001.

[7] Bertin J. *Sémiologie Graphique*. Gauthier-Villars, Paris, 1967.

[8] Briand LC, Daly JW, Wüst JK. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1): 91-121, January/February 1999.

[9] Briand LC, Emam KE, Morasca S. On the application of measurement theory in software engineering. *International Journal of Empirical Software Engineering*, 1(1): 61-88, October 1996.

[10] Briand LC, Wüst J. Empirical studies of quality models in object-oriented systems. To appear in Marvin Zelkowitz, editor, *Advances in Computers*, Academic Press, 2002.

[11] Boehm BW. Software engineering economics. *IEEE Transactions on Software Engineering*, 4(21), 1984.

[12] Boehm BW, Brown JR, Kaspar JR, and al. *Characteristics of Software Quality*. TRW Series of Software Technology. Amsterdam, North Holland, 1978.

[13] Boehm BW and al. *Software Cost Estimation with Cocomo II*. Prentice-Hall, January 2000.

[14] Budd T. *Object Oriented Programming*. Addison-Wesley, Reading, Massachusetts, 1991.

[15] Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, 1996.

[16] Card SK, Mackinlay J, Shneiderman B, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers, San Francisco, 1999.

[17] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6): 476-493, June 1994.

[18] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications* (OOPSLA'91), pages 197-211, 1991.

[19] Churcher NI, Sheppard MJ. Comments on *A metrics suite for object oriented design, IEEE Transactions on Software Engineering*, 21(3): 263-265, March 1995.

[20] Churcher NI, Sheppard MJ. Towards a conceptual framework for object oriented software metrics. *ACM SIGSOFT Software Engineering Notes*, 20(3): 69-76, April 1995.

[21] Demeyer S, Ducasse S, Lanza M. *A hybrid reverse engineering platform combining metrics and program visualization*. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[22] Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. In *Proceedings of OOPSLA'2000, ACM SIGPLAN Notices*, pages 166-178, 2000.

[23] Demeyer S, Ducasse S, Tichelaar S. Why unified is not universal - UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[24] Devanbu PT. *GENOA – a customizable, language and front-end independent code analyzer*. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 307-317. Melbourne, Australia. 1992.

[25] Discover. Documentation disponible en ligne : <http://www.mks.com/products/discover/>, novembre 2002.

[26] Dromey RG. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2): 146-162, 1995.

[27] Dufresne P, Robitaille S, Keller RK. SPOOL Design Browser documentation. *Technical Report GELO-127*, Université de Montréal, avril 2000.

[28] Eick SG, Graves TL, Karr AF, Mockus A, Schuster P. *Visualizing software changes. IEEE Transactions on Software Engineering*, 28(4): 396-412, 2002.

[29] FAMIX: Famoos information exchange. Documentation disponible en ligne : <http://www.iam.unibe.ch/~scg/Archive/famoos/FAMIX/>

[30] Fayad ME, Schmidt DC, Johnson RE, editors. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley and Sons, 1999.

[31] Fayad ME and Johnson RE, editors. *Domain-Specific Application Frameworks. Frameworks Experience by Industry*. John Wiley and Sons, 1999.

[32] Fenton N. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3): 199-206, March 1994.

[33] Fenton NE, Neil M. Software metrics: Roadmap. In *Proceedings of the Twenty-Second International Conference on Software Engineering*, pages 357-370, 2000.

[34] Fenton NE, Pfleeger SL. *Software Metrics:. A Rigorous and Practical Approach*. PWS Publishing Company, 1997.

[35] Foley JD, van Dam A, Feiner SK, Hughes JF. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., Reading, MA, 2nd edition in C,

[36] Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[37] Gamma E, Helm R, Johnson R, Vlissides J, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Menlo Park, CA. 1995.

[38] Gamma E, Weinand A. ET++: A Portable C++ Class Library for a UNIX Environment. Tutorial notes. *OOPSLA '90*. Ottawa, ON, Canada. October 1990.

[39] Healey CG. Choosing effective colours for data visualization. In *Proceedings of Visualization '96*, pp. 263–270, SanFrancisco, California, 1996.

[40] Healey CG, Enns JT. Large datasets at a glance: Combining textures and colors in scientific visualization. *IEEE Transactions on Visualization and Computer graphics*, 5(2): 145-167, 1999.

[41] Hitz M, Montazeri B. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4): 267-271, April 1996.

[42] IBM. *Object Constraint Language Specification 1.1*. Septembre 1997. Source :<http://www-3.ibm.com/software/ad/library/standards/ocl.html>

[43] ISO 9126. *Software engineering - product quality - part 1: Quality model*, June 2001. Source: <http://www.iso.ch>

[44] J2EE : Java 2 Entreprise Edition. Documentation disponible en ligne : <http://java.sun.com/j2ee>

[45] JClass Chart. Documentation disponible en ligne: <http://www.sitraka.com/software/jclass/>

[46] Kabaili H. Changeabilité des logiciels orientés objet : propriétés architecturales et indicateurs de qualité. PhD thesis, Université de Montréal, Montréal, Québec, Canada, January 2002.

[47] Keim DA. Visual exploration of large data sets. *Communications of the ACM*, 44, 2001.

[48] Keller RK, Schauer R, Robitaille S, Pagé P. Pattern-based reverse engineering of design components. In *Proceedings of the Twenty-First International Conference on Software Engineering*, pages 226-235, Los Angeles, CA, May 1999. IEEE.

[49] Kitchenham B, Pfleeger SL. Software quality: The elusive target. *IEEE Software*, 13(1): 12-21, January 1996.

[50] Krakatau Metrics Professional. Documentation disponible en ligne: <http://www.powersoftware.com/>

[51] Lanza M. *Combining metrics and graphs for object oriented reverse engineering*. Diploma thesis, University of Bern, October 1999.

[52] Lanza M. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, page to be published, 2001.

[53] Lanza M, Ducasse S. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300-311, 2001.

[54] Lanza M, Ducasse S, Steiger L. Understanding software evolution using a flexible query engine. In *Proceedings of the Workshop on Formal Foundations of Software Evolution*, 2001.

[55] Lewerentz C and Simon F. Integrating an object-oriented metrics tool into sniff+. *Technical Report I-22/1997*, University of Bern, 1997.

[56] Lorenz M, Kidd J. *Object-Oriented Software Metrics*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[57] Martel F. Documentation structurée du design des cadres d'application. Master's thesis, Université de Montréal, Montréal, Québec, Canada, October 1999.

[58] Mayrand J, Leblanc C, Merlo E. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the International Conference on Software Maintenance*, pp 244–253. Monterey, USA, November 1996.

[59] McCall JA, Richards PK, Walters GF. *Factors in software quality*. Technical Report RADC TR-77-369, Vols I, II, III, US Rome Air Development Center Reports NITS AD/A-049 014, 015, 055, 1977.

[60] MOOD: Metrics for Object Oriented Design. Documentation disponible en ligne : <http://www.esw.inesc-id.pt/ftp/pub/esw/mood/MoodPage/welcome.htm>

[61] NetBeans. Metamodel for Java Language. Documentation disponible en ligne : <http://java.netbeans.org/models/java/java-model.html>

[62] OMG. *Meta Object Facility(MOF) Specification* (version 1.4). Object Management Group, April 2002.
Source: <ftp://ftp.omg.org/pub/docs/formal/02-04-03.pdf>

[63] OMG. *OMG Unified Modeling Language Specification* (version 1.4). Object Management Group, September 2001.
Source: <ftp://ftp.omg.org/pub/docs/formal/01-09-67.pdf>

[64] OpenViz. Documentation disponible en ligne :
<http://www.avs.com/software/soft_b/openviz/index.html>

[65] POET 6.1. Documentation disponible en ligne :
<http://www.poet.com/>

[66] Pomerantz JR. Perceptual Organization in Information Processing. In Michael Kubovy and James Pomerantz editors, *Perceptual Organization*, chapter 6, pages 141-180. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981.

[67] Rational Rose. Documentation disponible en ligne:
<http://www.rational.com/products/rose/>

[68] Redwine S, Riddle W. Software technology maturation. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 189-200, May 1985.

[69] Roberts FS. *Measurement Theory with Applications to Decision making Utility and the Social Sciences*. Addison-Wesley, Reading, 1979.

[70] Robitaille S. *Support informatique à la compréhension des logiciels orientés objet de taille industrielle*. Master's thesis, Université de Montréal, Montréal, Québec, Canada, avril 2000.

[71] Robitaille S, Schauer R, Keller, RK, Bridging Program Comprehension Tools by Design Navigation. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, San Jose, CA, October 2000. IEEE.

[72] Schauer R, Keller RK, Laguë B, Knapen G, Robitaille S, Saint-Denis G. The SPOOL design repository: Architecture, schema, and mechanisms. In Hakan Erdogmus and Oryal Tanir, editors, *Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation*, pages 269-294. Springer, 2002.

[73] Shaw M. The coming-of-age of software architecture research. In *Proceedings of the Twenty-Third International Conference on Software Engineering*, pages 656-664a, 2001.

[74] Simon F, Steinbruckner F, Lewerentz C. Metrics based refactoring. In *CSMR*, pages 30-38, 2001.

[75] SNIFF+. Documentation disponible en ligne:
<http://www.windriver.com/products/sniff_plus/>

[76] Spool : Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems.
Information disponible en ligne:<http://www.iro.umontreal.ca/labs/gelo/spool/>

[77] St-Denis G. RCR: un profil UML pour la rétroconception, la compréhension et la réingénierie de logiciels. Master's thesis, Université de Montréal, Montréal, Québec, Canada, avril 2001.

[78] Stevens SS. On the theory of scales of measurement. *Science*, 103: 677-680, 1946.

[79] Stroulia E and Kapoor RV. Metrics of refactoring-based development: An experiment report. In *Proceedings of the Seventh International Conference on Object-Oriented Information Systems*, August 2001.

[80] Together ControlCenter. Documentation disponible en ligne : <http://www.togethersoft.com/>

[81] Treisman A, Gelade G. A feature-integration theory of attention. *Cognitive Psychology*, 12: 97-136, 1980.

[82] Treisman A, Gormican S. Feature analysis in early vision: Evidence from search asymmetries. *Psychological Review*, 95(1): 15-48, 1988.

[83] Unified Modeling Language. Documentation disponible en ligne : <http://www.omg.org/uml>

[84] Velleman P, Wilkinson L. Nominal, ordinal, interval, and ratio typologies are misleading. *The American Statistician*, 47(1): 65-72, February 1993.

[85] Ware C. Beatty JC. Using color dimensions to display data dimensions. *Human Factors*, 30, 1988.

[86] Ware C. Color sequences for univariate maps: Theory, experiments, and principles. *Human Factors*, 8(5): 41-49, 1988.

[87] Ware C. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[88] Web Gain Studio. Documentation disponible en ligne : <http://www.webgain.com/>

[89] Weyuker E. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9): 1357-1365, September 1988.

[90] XMI: XML Metadata Interchange. Documentation disponible en ligne: <http://www.omg.org/technology/documents/formal/xmi.htm>

[91] Yin R, Keller RK. Program Comprehension by Visualization in Contexts. In *Proceedings of the International Conference on Software Maintenance (ICSM'2002)*, pages 332-341, Montreal, Canada, October 2002. IEEE

[92] Zuse H. *A Framework of Software Measurement*. Walter de Gruyter, Berlin, New York, 1998.