MASTER EN SCIENCES INFORMATIQUES

Méthodologie de développement d'indicateurs de qualité orientés objet

Oprei, Pierre

Award date:
2006

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique

Méthodologie de développement d'indicateurs de qualité Orientés Objet

Pierre Oprei

Mémoire présenté en vue de l'obtention du grade de
Maître en Informatique

Année académique 2005 - 2006

Résumé

La recherche concernant l'évaluation des qualités logicielles d'un système étant riche et très pratiquée, nous tenterons d'apporter, via ce mémoire, des éléments méthodologiques permettant d'exploiter une nouvelle technique d'évaluation qualitative des logiciels orientés objet. Un système de visualisation en trois dimensions de logiciels, développé au laboratoire de Génie Logiciel de l'Université de Montréal, permet de soulever de nouvelles hypothèses qui ouvrent à leur tour de nouvelles pistes de recherche concernant l'évaluation qualitative d'attributs de la qualité logicielle, tels que la maintenabilité. C'est par cette nouvelle technique d'observation, permettant de mettre en évidence certaines propriétés orientées objet d'un logiciel, que nous sommes aujourd'hui capable d'évaluer rapidement, sans devoir analyser une multitude de valeurs numériques, certaines qualités orientées objet qui sont liées à la bonne utilisation ou non du paradigme Orienté Objet et de ses mécanismes. Une méthodologie et un plan de travail ont été suivis pour parvenir à l'élaboration de nouveaux indicateurs qualitatifs de certains attributs de la qualité logicielle orientée objet, liés au système de visualisation utilisé. Une hypothèse secondaire a été exploitée et des premiers résultats ont été obtenus. Cette dernière pousse à penser que la complexité cyclomatique de MacCabe pourrait être, à elle seule et observée avec l'outil de visualisation et ses techniques, un indicateur de bonne ou de mauvaise utilisation de l'Orienté Objet et de ses principes.

Abstract

Considering that the research domain which concerns software quality evaluation is very popular, we will try to bring, through this document, methodological elements to give the opportunity to work with a new technique of qualitative software evaluation, specifically for object-oriented softwares. A new tool, developed in the Software Engineering Laboratory of the University of Montreal, offers the possibility of visualizing software in three dimensions. This tool brings forward new opportunities to develop hypotheses which will lead to new axes of research in the field of qualitative evaluation of software quality attributes, such as the qualitative evaluation of maintainability. Thanks to this new observation technique, which offers the necessary tools to highlight some object-oriented attributes, we are able to rapidly evaluate object-oriented qualities linked to the right or wrong uses of the Object-Oriented paradigm, and of its mechanisms, without any analysis of complex numerical values. We followed a strict methodology and a plan to complete the elaboration of new qualitative indicators which give information about certain quality attributes of object-oriented software quality. These indicators have to be used with the new visualization tool. We followed a secondary hypothesis and we obtained some interesting results. This last idea pushes us to think that the cyclomatic complexity of MacCabe can be an indicator of right or wrong uses of the Object-Oriented and of its mechanisms once it is observed alone and with our new visualization tool.

Avant-propos

Ce document est l'aboutissement d'une recherche débutée à l'Université de Montréal (UdeM), Canada. Ce travail a été entrepris dans le cadre d'un stage de fin d'études universitaires à l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix (FUNDP) de Namur, Belgique. Les résultats obtenus sont le fruit des efforts fournis par différentes personnes, membres de cette collaboration entre ces deux universités.

Ce document n'a évidemment pas été rédigé par un seul étudiant mais il est aussi l'oeuvre de plusieurs personnes qui ont apporté leur aide, leurs conseils et leur savoir-faire. Je désire donc remercier tous ceux qui, de près ou de loin, ont permis la réalisation de ce travail.

Je remercie tout d'abord le Professeur Naji Habra, promotteur de ce mémoire, qui m'a apporté conseils et encouragements. Je remercie également le Professeur Houari Sahraoui qui m'a accueilli au sein de son équipe du laboratoire de Génie Logiciel de l'Université de Montréal, et grâce à qui j'ai pu réaliser une grande partie de mes recherches. Par la même occasion, je remercie tous les membres de ce laboratoire qui ont pu me conseiller et m'aider dans mon travail.

Je remercie aussi, bien entendu, tous mes proches pour leurs encouragements et leur patience.

Table des matières

Table des figures

Première partie

Contexte

Chapitre 1

Introduction

Ce mémoire est le fruit d'une réflexion et de recherches entreprises dans le cadre d'un stage de dernière année de Maîtrise en Informatique à l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix de Namur (FUNDP).

Cette réflexion baigne dans le domaine du génie logiciel et plus particulièrement dans le courant des recherches et des travaux qui s'intéressent aux qualités logicielles d'un système orienté objet. Le travail qui sera présenté dans ce document aborde une nouvelle possibilité d'évaluation des qualités d'un logiciel orienté objet ainsi que les premiers résultats obtenus en exploitant cette nouvelle méthode.

Ce document abordera les résultats obtenus après avoir situé le contexte général et la méthodologie suivie. Le contenu est divisé en trois grandes parties: la première situe le cadre et le contexte du mémoire, la seconde présente le travail réalisé et la troisième consiste en une annexe.

1.1 La crise du logiciel et la prise de conscience vis-à-vis de la qualité logicielle

Il est dans la nature de l'homme de ressentir la nécessité de faire preuve de réflexion envers lui-même et envers ses accomplissements. Tout au long de son histoire, l'homme a fait de grandes découvertes et de grands progrès techniques, et à chaque fois, quelques érudits se sont penchés sur les questions qui entouraient ces progrès. De nos jours, un grand nombre d'espoirs de progrès de l'humanité repose sur la branche scientifique informatique. Cette recherche et ces évolutions, sans cesse plus rapides et fréquentes, ont également remis en question la qualité et la nécessité de ce progrès. Depuis un certain nombre d'années, la branche informatique de génie logiciel joue un peu le rôle de réflexion et de critique envers les nouvelles tendances de développement et envers les nouvelles technologies logicielles. De grands centres de recherche se sont penchés sur les différentes questions qui entouraient le développement massif de logiciels.

Cette prise de recul de la part des laboratoires de génie logiciel s'est avérée nécessaire quand le monde de l'informatique s'est aperçu que le développement de logiciels devenait de plus en plus coûteux et surtout de moins en moins efficace. En effet, la plupart des programmes développés devenaient rapidement inutilisables, entre autres à cause de leur trop grande difficulté de maintenance et de modification. C'est ce qu'on appelle la *crise du logiciel* [STR00], identifiée et citée pour la première fois dans les travaux de Barry W. Boehm [BOE73]. Cette crise a été détectée suite à plusieurs observations:

- les logiciels développés correspondaient rarement aux attentes et aux exigences des clients, des utilisateurs,
- les logiciels contenaient beaucoup trop d'erreurs,
- la maintenance des logiciels etait devenue une tâche trop coûteuse et trop complexe,
- les coûts de développement de logiciels étaient difficilement prévisibles et les sommes allouées atteignaient vite des montants prohibitifs,
- les délais de développement et de livraison de logiciels étaient presque toujours dépassés,
- les logiciels développés étaient très rarement portables.

Tous ces symptômes ont évidemment entraîné une perte de confiance de la part des utilisateurs et des clients envers les développeurs de logiciels et envers la *solution informatique* que tout le monde conseillait. Un esprit plus critique s'avérait donc nécessaire tout au long du processus de développement d'un logiciel pour amener des solutions à cette crise. C'est ainsi que nous avons vu les premières suites de mesures apparaître, ou encore d'autres outils de quantification tels que ceux permettant le calcul de la complexité. Ces outils devaient permettre d'évaluer la qualité d'un logiciel tout au long de son développement.

1.2 La question de la qualité logicielle

Une des réponses à la crise du logiciel a été de revenir à des concepts plus généraux de qualité qu'on retrouvait dans d'autres disciplines. Ces principes ont été appliqués au développement de logiciels.

La qualité est évidemment une notion très subjective qui fait surgir un grand nombre de concepts tels que la rapidité, le confort, les erreurs, la facilité, ... Le concept de qualité, en général, a beaucoup évolué au cours du temps. Il est passé du simple contrôle de qualité après production à la gestion et au développement *par* la qualité, c'est à dire un suivi de qualité tout au long de la production d'un produit. On observe donc une prise de conscience globale face à ce problème. C'est d'ailleurs grâce à cette évolution que nous avons vu apparaître de nombreuses normes de qualité mises en place par les entreprises elles-même, soucieuses de la reconnaissance de la valeur qualitative de leur produit face à celui de la concurrence. Cette notion de qualité est aujourd'hui presque indissociable de

la notion de concurrence et de reconnaissance.

Nous observons évidemment la même tendance au sein du développement logiciel. Suite à la crise qui a frappé le développement de logiciels, il était nécessaire de s'assurer de la bonne qualité d'un système développé. Cette notion de qualité pour un logiciel se rapproche évidemment de la notion générale mais est aussi caractérisée par un grand nombre d'attributs de qualité très variés et parfois encore méconnus. On parle souvent de maintenabilité d'un logiciel en parallèle avec le concept de qualité logicielle. Les utilisateurs désirent donc actuellement un logiciel qui ne contient pas d'erreur, qui est développé dans les temps et pour un budget respecté. Le produit doit aussi satisfaire les attentes des utilisateurs et être facilement adaptable et modifiable dans le futur. Les deux grands pôles sont donc la gestion des erreurs d'un système et le maintien de sa facilité de maintenance et de modification. Il est évident que nous pouvons nous rapporter également à des modèles de qualité logicielle, tels que celui proposé par Barry W. Boehm [BOE78] (voir figure 1.1). La maintenabilité est ici un attribut de qualité à part entière. On peut distinguer des attributs de qualité du logiciel tel quel, mais aussi des attributs de qualité du logiciel en évolution, comme la maintenabilité ou la facilité de modification.

Le besoin de pouvoir mesurer la qualité d'un système est devenu de plus en plus nécessaire, entre autres pour assurer une facilité de maintenance et la satisfaction du client. Pourquoi la maintenance? Parce qu'aujourd'hui rares sont les systèmes qui, même s'ils sont vendus en tant que versions définitives, ne subissent pas des modifications tout au long de leur utilisation. Ce concept est donc au centre même du débat concernant la qualité logicielle.

1.3 La qualité logicielle et les systèmes orientés objet

Le besoin de pouvoir mesurer les qualités en général et la maintenabilité en particulier d'un système est devenu de plus en plus nécessaire. Si on restreint ce contexte à celui des logiciels orientés objet, nous remarquons que ces dernières années ont été dominées par de grands courants se disant être "la bonne façon de programmer en Orienté Objet" ainsi que par la promotion de l'utilisation des patrons de conception (*design patterns*) [GHJV94]. Cet exemple montre bien la nécessité d'atteindre une certaine qualité reconnue par tous à l'issue du développement d'un projet informatique. Mais il ne faut pas confondre méthode et solution universelle. L'Orienté Objet se présente comme un paradigme qui facilite l'évolution d'un logiciel, mais il est évident que ce paradigme doit être bien employé pour bénéficier de cette caractéristique. Les *design patterns*, par exemple, ne constituent qu'un mode d'emploi regroupant des solutions efficaces et reconnues à des problèmes souvent rencontrés, ils ne constituent pas vraiment un guide complet de qualité logicielle. Par ailleurs, une utilisation abusive de ces *best practices* peut évidemment rendre la compréhension de l'architecture ardue et donc la tâche de maintenance du logiciel extrêmement compliquée.

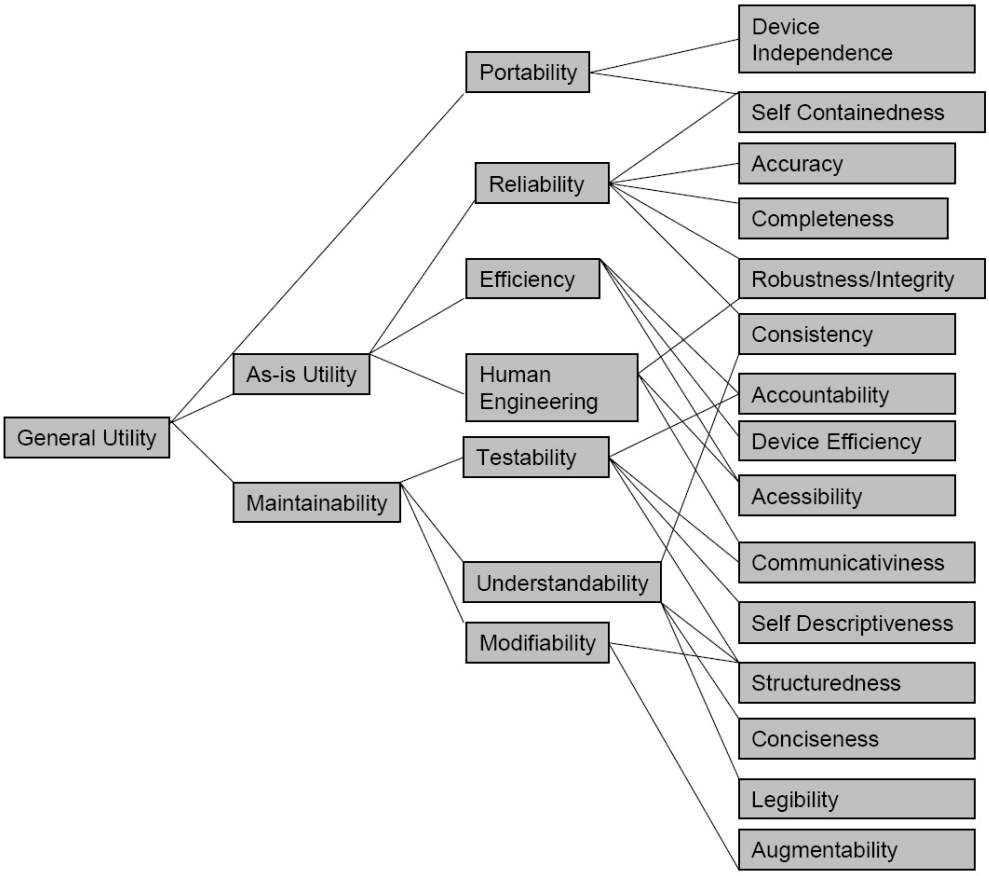

FIG. 1.1 – *Arbre des caractéristiques de qualité d'un logiciel selon Barry W. Boehm.*

En ce qui concerne l'analyse des qualités mêmes d'un logiciel orienté objet, nous pouvons remarquer l'énorme quantité de publications qui couvrent le sujet des mesures logicielles adaptées aux systèmes orientés objet [BBM96]. Ces mesures forment aujourd'hui une sorte de librairie de tests de qualité. Ces calculs de valeurs sont également utilisés pour améliorer un logiciel, d'un point de vue design [EL96] [TK03], ou dans un but de refactorisation (*refactoring*) [SSL01] [FOW].

Il est donc clair que le problème de qualité logicielle touche les systèmes orientés objet. Nous verrons plus tard dans la section 2.2 (page 21) de quelle façon il est possible d'aborder ce problème. Il existe en effet plusieurs écoles et nous fairons un point de la situation.

Ce mémoire s'inscrit donc dans cette mouvance qu'est celle de l'analyse de logiciels. La recherche menée ici consiste en une recherche de qualité par la création d'indicateurs permettant de juger de la qualité d'un logiciel orienté objet dans le but, entre autres, d'améliorer la maintenabilité d'un système. Dans notre réflexion, nous aborderons les problèmes de la qualité logicielle et en particulier de la maintenabilité des logiciels orientés objet. La réflexion générale se base sur l'utilisation de plusieurs outils définis ci-dessous, tels que des mesures logicielles orientées objet. La nouveauté apportée ici sera la façon d'utiliser et d'observer ces différents indicateurs. En effet, nous montrerons comment avoir une idée claire d'un très grand nombre de valeurs différentes grâce à une simple observation visuelle. La recherche menée sur cette nouvelle manière d'observer certaines caractéristiques quantifiées d'un logiciel nous a permis de suivre plusieurs pistes.

Ce document poursuivra deux objectifs, un général et un secondaire. Comme nous le verrons plus tard, l'outil d'observation utilisé est encore très peu documenté et son utilisation n'est pas encore publique. Nous tenterons donc, dans un premier temps, de définir les possibilités de ce logiciel. Pour ce faire, nous élaborerons de nouveaux indicateurs basés à la fois sur les nouvelles opportunités d'observation qu'offre cet outil, ainsi que sur certaines mesures logicielles bien connues. Ces indicateurs devraient nous permettre de détecter, avec un certain niveau de fiabilité, la bonne utilisation ou non des principes orientés objet. Dans un deuxième temps, nous expliquerons pourquoi et comment nous avons orienté notre recherche selon une intuition secondaire. Nous avons eu l'idée que l'observation de la complexité cyclomatique au niveau du système entier (sans observer classe par classe) pouvait à elle seule nous indiquer le "degré d'orientation objet" d'un système. Une partie de la recherche s'est donc naturellement orientée vers cet objectif. Toutes les notions abordées ici restent bien sûr à définir.

Pour atteindre ces deux objectifs, nous diviserons le document en plusieurs grandes parties. La structure se divise comme suit: la première partie fixe le cadre du mémoire en présentant l'état de l'art ainsi que les différents outils utilisés; la deuxième partie explique comment la recherche a été menée, quelle méthodologie a été suivie, et présente les objectifs qui ont été réalisés ainsi que les résultats obtenus. La dernière partie contient

les annexes nécessaires.

Chapitre 2

État de l'art en qualité et mesures orientées objet

2.1 Vocabulaire

Tout au long de ce document, nous allons employer plusieurs termes qu'il est nécessaire de définir pour ne pas semer la confusion. Certaines de ces notions sont tirées de la littérature concernant le génie logiciel en général [HALS06].

- **Entité:** Ce terme renvoit à un objet pour lequel une mesure peut être prise. Une *entité* peut être par exemple un morceau de code, un partie de l'architecture, un tâche d'exécution, un processus de maintenance, ... Dans notre cas, l'*entité* sera souvent une classe du programme orienté objet (spécifiquement Java). Une *entité* est caractérisée par un ensemble d'attributs, qui correspondent chacun à une propriété simple observable. Nous appellerons aussi ces propriétés des *propriétés de classe*.
- **Attribut:** L'*attribut* est la propriété d'une *entité* qui peut être déterminée quantitativement, autrement dit, à laquelle nous pouvons assigner une grandeur. Nous pouvons distinguer les *attributs* de base et les dérivés. Un *attribut* de base d'une classe peut être par exemple le nombre de méthodes qu'elle contient. Et un *attribut* dérivé sera par exemple la complexité cyclomatique de la classe, qui est en fait une combinaison des complexités cyclomatiques de chacune des méthodes. En général, nous appellerons *attribut* des caractéristiques comme le couplage d'une classe vis-à-vis des autres ou encore la cohésion de celle-ci.
- **Propriété de classe:** Comme nous le verrons plus tard, l'outil de visualisation qui a été utilisé permet de représenter des classes de logiciel orienté objet en trois dimensions. Nous avons présenté le terme *attribut* mais dans notre développement nous parlerons plus souvent de *propriétés de classe*. Ces propriétés sont en effet bien souvent des *attributs* dérivés d'une *entité* qui est la classe elle-même. Quand nous parlerons d'une *propriété de classe*, *propriété* prendra plus précisément le sens de *métrique*, ce terme est défini ci-après. Les principales *propriétés de classe* qui seront observées seront des valeurs de CBO, DIT, LCOM5, et WMCC (voir section 3.1.2

page 34 pour plus de détails).

- **Mesure (d'un attribut):** La *mesure* d'un attribut est la caractérisation de celui-ci en terme de nombres ou de symboles. Le processus de mesure est en fait l'application d'une entité empirique à un nombre. Plus précisément, la *mesure* d'un *attribut* d'une *entité* est le processus permettant d'obtenir des informations numériques à propos de la grandeur de cet *attribut*. Il est évident qu'il faut aussi parler du *résultat de la mesure* car bien souvent le terme *mesure* désigne en fait le résultat, de la même façon que le terme *métrique* désigne souvent la valeur obtenue suite au calcul de la *métrique* en question.
- **Métrique:** Dans la littérature, le terme *métrique* est de plus en plus évité car il est ambigu. Donc, pour éviter toute confusion, quand nous parlerons de *métrique*, il s'agira automatiquement d'une *métrique* faisant partie d'une suite d'outils permettant d'évaluer certains attributs d'un logiciel. Le terme *métrique* est évidemment proche de celui de *mesure* et c'est pour cette raison que nous n'appellerons *métrique* que seules les mesures qui ont été définies comme telles par leur(s) concepteur(s).
- **Indicateur:** Ce terme est encore très proche de celui de *mesure* ou de *métrique*. Il est évident que certaines *métriques* ou *mesures* peuvent s'avérer être des *indicateurs*. La différence se situe au niveau du processus de "mapping". En effet la mesure est l'application d'une grandeur mesurable (*attribut*) à une valeur numérique ou symbolique, alors qu'un indicateur est une valeur numérique ou autre, qui nous permet de tirer des conclusions. Mais cette valeur n'est pas nécessairement directement le résultat de la mesure d'un attribut. Si c'est une valeur numérique, l'indicateur peut par exemple être une combinaison linéaire de plusieurs mesures. Dans notre cas, nous parlerons donc d'*indicateur* car nous tenterons de créer un outil permettant d'indiquer si un code utilise de façon efficace les principes orientés objet en vue de faciliter la maintenance. Cet indicateur sera composé grâce à l'observation du résultat de plusieurs métriques à une échelle de système entier et aussi grâce à l'observation de propriétés plus générales.
- **Propriété de design:** En parlant de *propriétés du design*, nous nous situons à un niveau plus haut que quand nous parlons d'*attribut* ou de *propriétés de classe*. Cette notion sera employée surtout lors d'observation avec l'outil de visualisation. Quand on analyse un système entier (numériquement ou, comme dans ce cas précis, visuellement), on peut critiquer des caractéristiques telles que *l'équilibrage du couplage ou de la complexité dans un package*. Il est également possible de critiquer l'utilisation du mécanisme d'héritage: *forte utilisation, faible utilisation, ou encore utilisation isolée et extrême*. Nous appellerons donc ces caractéristiques plus générales, qui sont observables à un niveau plus élevé que celui de la classe isolée, des *propriétés du design*.
- **Types de mauvaise utilisation de l'orienté objet:** Il existe plusieurs *types* de mauvaise utilisation des principes orientés objet que nous pouvons facilement détecter grâce à l'outil de visualisation. Avec les *propriétés de classe*, nous avons vu que nous pouvions déduire des *propriétés de design*, en observant ces *propriétés de classe* en générale, c'est à dire en analysant plusieurs classes à la fois et non pas une

isolément. Une fois que nous avons remarqué plusieurs propriétés du design, nous pouvons conclure qu'un système se classe dans un ou plusieurs *types* de mauvaise utilisation des principes orientés objet. Les différents *types* de défauts se situent au niveau du couplage du système, de la cohésion de celui-ci et de l'utilisation de l'héritage qui est fait au sein de l'architecture. Nous verrons plus en détails ces *types* de défauts dans la section "Elaboration des indicateurs visuels". C'est grâce à cette classification d'un système dans un ou plusieurs *types* de mauvaise utilisation de l'Orienté Objet que nous pourrons ensuite tirer des conclusions concernant les qualités orientées objet du système, et plus précisément sur sa maintenabilité.

2.2 Qualité des logiciels orientés objet

L'Orienté Objet se présente souvent comme LA solution aux différents problèmes rencontrés en développement de logiciels. Il est vrai que l'Orienté Objet a déjà fait ses preuves d'efficacité pour des systèmes qui doivent être robustes et facilement maintenables. Seulement, alors que le design orienté objet est caractérisé de robuste, maintenable et réutilisable, ce n'est pas pour autant que tous les systèmes codés en Orienté Objet jouissent de ces propriétés, ou en tous cas pas au même degré. Nous pouvons donc conclure que ces caractéristiques ne sont pas intrinsèques au paradigme Orienté Objet. Il est d'ailleurs possible de montrer qu'il est nécessaire de s'assurer qu'une certaine qualité est maintenue tout au long du développement pour s'assurer que le système codé offrira une grande réutilisabilité tout en étant robuste et facilement modifiable [MAR95]. Cette qualité de développement est comparable au "bon usage" de l'Orienté Objet pour bien tirer profit de ce que ce paradigme peut offrir.

Dans cette section seront présentées les différentes familles d'approches traitant de la qualité des logiciels orientés objet. On recense principalement deux axes: celui de l'approche basée sur des méthodes qualitatives et celui de l'approche basée sur des techniques quantitatives.

2.2.1 Approches qualitatives

Les approches qualitatives peuvent être classées selon deux grandes catégories: celle qui reprend les bonnes pratiques d'utilisation de l'Orienté Objet et celle qui reprend les mauvais usages typiques des principes orientés objet. Cette dernière catégorie se subdivise encore en deux sous-classes. La première reprend les habituelles mauvaises solutions au niveau architectural alors que la deuxième reprend plutôt les mauvaises habitudes de codage. Mais il faut préciser que ce qui est reconnu comme bonne solution ou bonne pratique d'utilisation orientée objet n'est rien d'autre que ce qui est accepté, admis et reconnu par un consensus d'experts et d'utilisateurs. Il n'y a donc aucune évidence empirique.

Pour commencer, nous allons parler de la branche la plus exploitée actuellement. Il s'agit bien sûr de tout ce qui touche les *best practices* de programmation orientée objet,

ou en tous cas ce qu'on peut qualifier de "bonnes pratiques", non définies clairement dans la littérature mais généralement acceptées et admises par un consensus large d'experts en développement logiciel. En particulier nous pouvons citer la promotion des *design patterns* [GHJV94], les recherches suivies sur la détection de *bad smells* dans le code [MÄN03] ou encore la détection d'anti-patrons de conception (*anti-patterns*) [MG05] [MHG06].

La première technique consiste à donner des solutions d'architecture, de design à des familles de problèmes connus de programmation, citons par exemple le *pattern* dit du *Visiteur*. Cette façon d'architecturer les classes entre elles a pour objectif de réduire le couplage entre la structure de données et les actions sur celles-ci, ce qui nous permet de spécifier des algorithmes à l'extérieur des structures de données sur lesquelles ils s'appliquent. La question qui se pose est: "En quoi l'utilisation de *design patterns*, ou dans quelle mesure cette utilisation garantit une certaine qualité du logiciel?". Comme nous l'avons dit dans la section précédente, la qualité logicielle orientée objet est caractérisée entre autres par le fait que le logiciel soit facilement maintenable et réutilisable tout en étant robuste. Or, les solutions proposées par la panoplie de patrons de conception présente dans la littérature sont justement des solutions d'architecture reconnues par une majeure partie de la communauté des développeurs. Et, de plus, les techniques de programmation associées facilitent énormément la réutilisation vu que ces différentes architectures sont énormément documentées et sont connues par les développeurs partisans de l'utilisation des *design patterns* orientés objet. Par ailleurs, la maintenance et les autres types de modifications sont également facilitées car la plupart des *design patterns* ont été conçus dans un souci de facilité d'adaptation et de modification. Nous qualifierons cette approche de qualitative vu qu'elle se base sur des *best practices* de programmation reconnues. On considère qu'un système qui est programmé grâce à l'utilisation de *design patterns* reconnus est un système qui fait preuve d'une certaine qualité au sens orienté objet du terme. Mais, de nouveau, aucune évidence ne peut être trouvée dans la littérature, il s'agit de ce qui est habituellement reconnu et accepté. Aussi, répétons le encore une fois, seule l'utilisation modérée et réfléchie des *design patterns* peut aboutir à l'assurance d'une certaine qualité.

Le seconde méthode qualifiée de qualitative que nous pouvons brièvement présenter est celle de détection de *bad smells*. Les *code bad smells*, selon Martin Fowler et Kent Beck [MÄN03], sont des structures dans le code sources des programmes qui réduisent la maintenabilité des logiciels. Et comme nous l'avons déjà dit, cette facilité de maintenance influence grandement les coûts qui seront alloués dans la dernière partie du développement d'un logiciel. Contrairement à la technique précédente, celle-ci se situe au niveau du code et non plus au niveau de l'architecture. Nous pouvons citer quelques mauvaises habitudes de programmation qualifiée de *bad smells*:

- l'abus de très longues méthodes et de très longues listes de paramètres: il est souvent plus facile de comprendre et de modifier plusieurs petites méthodes imbriquées plutôt qu'une seule très longue,
- l'abus de principes orientés objet comme des arbres d'héritages parallèles excessivement profonds,

– l'abus de *data Class*, la duplication de code, ... En résumé, tout ce qui représente du code qui s'avère inutile et qui devrait être enlevé.

Il existe évidemment une taxonomie bien plus complète des *bad smells* reconnus [MÄN03]. Leur détection se fait par analyse simple du code ou plus efficacement par détection automatique, notamment grâce à des métriques orientées objet que nous aborderons dans la section suivante.

En ce qui concerne la détection de défauts dans un code, il existe une autre école qui est celle de la détection des *anti-patterns*. A l'instar des *code bad smells*, les *anti-patterns* sont aussi des défauts qui doivent être corrigés pour améliorer la qualité du logiciel et pour réduire les coûts de sa maintenance. Nous travaillons une fois de plus au niveau architectural dans le cadre de cette méthode. De la même manière que les *design patterns* sont des bonnes solutions pour des problèmes de développement connus, les *anti-patterns* sont des mauvaises habitudes, des mauvaises solutions à des problèmes souvent rencontrés. Certains *anti-patterns* sont reconnus et malheureusement trop souvent rencontrés. Citons l'exemple du *Blob*. qui est identifié par une classe qui monopolise tout le traitement et qui encapsule toutes les données. Les symptômes sont:

– une classe avec un grand nombre d'attributs et d'opérations,
– cette classe communique avec des classes de données,
– cohésion faible dans cette classe,
– absence de conception orientée objet.

Ce défaut architectural de conception engendre évidemment des conséquences: difficulté de tester cette classe, de la réutiliser, de la modifier sans devoir changer beaucoup d'autres composants,... La détection des *anti-patterns* peut se faire de plusieurs façons: analyse des commentaires dans le code, analyse du comportement au regard des diagrammes de séquences recréés à partir de l'exécution du code, ou encore, comme pour la détection de *code bad smells*, l'utilisation de mesures orientées objet pour, par exemple, faire l'analyse de la cohésion des classes. Cette école se rapproche beaucoup de l'optique dans laquelle a été menée cette recherche. Dans ce cas-ci, il sera également question de détection de défauts de conception grâce notamment à des mesures orientées objet. Nous ne serons pas tout à fait dans le cas de la détection d'*anti-patterns*, mais nous nous baserons aussi sur l'architecture générale du système pour critiquer la qualité de celui-ci.

Nous avons vu que certaines des techniques qualitatives se basent également sur l'utilisation de calculs de valeurs. Nous allons maintenant présenter ces différentes méthodes dites quantitatives.

2.2.2 Approches quantitatives

Ici, nous allons présenter comment des techniques quantitatives peuvent nous permettre d'aborder le problème de la qualité logicielle. Suite à la crise du logiciel, il s'est

avéré nécessaire de pouvoir mesurer et d'évaluer la qualité d'un logiciel pour réduire ses coûts de développement. Nous aborderons principalement les différentes méthodes qui utilisent des calculs de mesures pour parvenir à des conclusions de bonne ou mauvaise qualité. Les mesures sont évidemment indispensables pour répondre à une multitude de questions. On peut par exemple connaître l'impact d'une prise de décision, la fréquence de certains évènements tels que certaines erreurs, combien va coûter la maintenance du logiciel, le pourcentage des spécifications qui sont respectées, ...

Multitude de mesures

Il est évident que nous pouvons citer une énorme quantité de mesures. Les mesures logicielles vont du calcul simple du nombre de lignes de codes à des calculs très complexes d'interdépendances entre modules, en passant bien sûr par les différents calculs de complexité qui ont été proposés. Il existe plusieurs catégories de mesures: celles qui estiment des valeurs de comportement (nombre d'interactions avec l'utilisateur par exemple), les mesures se basant sur l'architecture, les mesures orientées objet, les mesures d'estimation de délai et de coût et encore bien d'autres telles que les mesures pour systèmes distribués.

La méthode la plus connue est évidemment l'utilisation des mesures de caractéristiques orientées objet qu'on appelle métriques. Les métriques orientées objet rendent des valeurs caractéristiques de certaines propriétés du design orienté objet. Nous verrons dans la section 2.3 (page 26) quelles boîtes à outils existent pour l'utilisation de ces mesures orientées objet. Dans le contexte de notre travail, nous nous pencherons plus en détails sur certaines de ces métriques que nous utiliserons pour critiquer un logiciel.

Lien avec la qualité et la maintenabilité

Comme nous l'avons dit précédemment, certaines méthodes de détection de défauts dans le code ou de défauts de conception se basent sur des valeurs calculées grâce à des métriques orientées objet ou autres mesures logicielles. Mais il est également possible d'utiliser des mesures pour détecter une mauvaise qualité de logiciel ou une future difficulté de maintenance sans devoir s'inscrire dans une approche de détection formelle d'*anti-patterns* ou de *code bad smells*. Les métriques connues pour les systèmes orientés objets permettent par exemple d'évaluer le couplage inter-classes d'un système sur base du code source. Ces interdépendances permettent de savoir quelles parties du système sont susceptibles d'être modifiées en cas de modification d'une autre partie du code. Nous pouvons également citer tous les calculs de complexité permettant d'indiquer la difficulté de maintenance du système par exemple.

Pour conclure, nous pouvons dire qu'il existe une multitude de mesures et d'utilisations de celles-ci. Les mesures logicielles peuvent être aujourd'hui utilisées tout au long du développement d'un logiciel pour permettre d'évaluer presque tous les aspects du futur système et de son développement. Cette approche est une approche de développement *par* la qualité, c'est-à-dire qu'il y a un contrôle permanent tout au long du processus

permettant d'anticiper et de prendre de meilleures décisions. Malgré ce constat, il est difficile d'établir un lien clair entre des mesures et la maintenabilité ou d'autres attributs de qualité d'un système.

2.2.3 Utilisation d'indicateurs

C'est dans cette section qu'il nous sera possible d'établir un lien entre des mesures quantitatives et les qualités d'un système. En effet, une mesure est un lien entre un attribut et une valeur numérique alors qu'un indicateur est une valeur, un graphique, ... qui nous aide à tirer des conclusions. Si c'est une valeur par exemple, celle-ci n'est pas nécessairement le *mapping* d'un attribut (dans le sens de "propriété d'une entité") qu'on veut mesurer sur une valeur numérique. Celle-ci est peut être composée d'une combinaison linéaire de plusieurs mesures.

Comme cité précédemment, l'idée principale de cette recherche était de pouvoir créer un indicateur montrant si un système orienté objet était un "bon" programme orienté objet ou non. Plus précisément, un programme objet respecte-t-il et utilise-t-il correctement les principes de la programmation orienté-objet? Dans cette section, nous allons donc exposer ce qui existe et ce qui se fait à l'heure actuelle dans ce domaine pour pouvoir inscrire cette recherche dans un courant existant.

Dans le monde du génie logiciel, la recherche sur les métriques orientées objet est une branche très exploitée. Un grand nombre de chercheurs tentent de définir des normes de qualité logicielle pour les programmes orientés objet en se basant sur la mesure de différents attributs d'un code (polymorphisme, complexité, héritage, couplage, cohésion, ...). Dans cette optique-là, les chercheurs tentent de valider ces métriques en tant qu'indicateurs de qualité ou d'indicateurs de défauts. On peut par exemple utiliser des mesures de cohésion d'une classe pour décider si un programme est de bonne qualité ou non. En effet, on considère qu'un système composé d'une grande majorité de classes non cohésives sera difficilement modifiable et réutilisable. Dans ce cas, on conclut donc que la qualité de ce système orienté objet peut laisser à désirer.

En terme d'indicateur d'"orientation objet" et d'indicateur de qualité en général, certains ont essayé de définir une méthode pour pouvoir affirmer qu'un programme est plus orienté objet qu'un autre [MAR03]. Le but de cette recherche est de savoir s'il était possible de mesurer l'"orientation objet" d'un système. La conclusion est qu'il est possible de mesurer le taux d'"orientation objet" de la totalité du code d'un système, c'est à dire le pourcentage de code dit "orienté objet" (selon la définition donnée dans le cadre de cette recherche) par rapport à tout le reste du code du logiciel. Cet indicateur est basé sur une multitude de mesures dont les valeurs sont calculées. Ces mesures sont ensuite analysées pour arriver à une conclusion combinée. Cet indicateur est donc capable de critiquer la qualité d'un système entier plutôt que de faire une simple analyse d'une classe à la fois, comme le font la plupart des métriques orientées objet. Cette approche est une

des seules permettant la critique et l'évaluation d'un système orienté objet entier de façon quantifiable, en terme de degré d'"orientation objet". Mais cela ne nous mène pas encore tout à fait à la qualité même d'un logiciel orienté objet. En effet, le fait d'avoir un plus grand pourcentage de code dit orienté objet ne qualifie pas directement un système de bon système orienté objet. Comme nous l'avons dit, le fait d'utiliser énormément les paradigmes orientés objets (polymorphisme, héritage,...) ne garantit pas nécessairement une bonne qualité, au sens de facilité de maintenance par exemple. L'abus d'utilisation du polymorphisme et de l'héritage peut rendre la structure difficilement compréhensible et donc difficilement modifiable.

Comme nous pouvons le constater, il n'existe pas encore à l'heure actuelle d'indicateur orienté objet simple et efficace nous permettant de séparer les logiciels bien programmés de ceux qui ne le sont pas, dans le sens où un système respecte bien les principes orientés objet, ce qui garantit une certaine robustesse et une facilité de maintenance et de réutilisation. De plus, les différentes méthodes existantes pour repérer les défauts d'un code ou tout simplement pour indiquer si un système pourrait être difficilement maintenable ou non se basent sur des analyses chiffrées souvent sur la base de métriques orientées objet. Dans notre cas, il s'agira de percevoir visuellement les bonnes ou mauvaises caractéristiques d'un système, même si les métriques orientées objet seront au coeur de notre réflexion.

2.3 Mesures Orientées Objet existantes

Il ne s'agit pas ici de donner une définition de ce qu'est une mesure orientée objet mais plutôt de faire un tour d'horizon des suites de mesures qui existent dans le monde de l'Orienté Objet et qui sont couramment utilisées. Notre choix se portera également sur certaines d'entre elles.

2.3.1 Suite de Chidamber et Kemerer

Une des suites de métriques orientées objet les plus connues est celle proposée par Chidamber et Kemerer [CK94]. Cette suite a été développée pour accompagner le développement de logiciel. Beaucoup de métriques existaient déjà mais celles-ci permettaient d'évaluer les systèmes orientés objet dont la popularité augmentait sans cesse. Cette suite orientée objet est composée de 6 métriques:

- **WMC (Weighted Method per Class):** WMC est une mesure de complexité. En considérant que les méthodes sont des propriétés des classes et que la complexité est déterminée par la cardinalité de l'ensemble de ses propriétés, alors WMC est la somme des complexités de chaque méthode. Cette métrique est très souvent utilisée en fixant la complexité de chaque méthode à la valeur unité. Plusieurs considérations peuvent être faites concernant cette métrique:

  1. le nombre de méthodes et leur complexité peuvent être utilisés pour prédire

le coût en effort qui est nécessaire pour développer cette classe et pour la maintenir,
2. plus le nombre de méthodes sera grand, plus l'impact sur les enfants (héritiers) pourrait être grand, vu que ceux-ci vont hériter de toutes ces méthodes,
3. les classes contenant un grand nombre de méthodes sont souvent qualifiées de spécialisées pour une application, ce qui limite leur réutilisation.

– **DIT (Depth of Inheritance Tree):** Cette métrique calcule la distance entre la racine de l'arbre d'héritage et la classe demandée. En cas de multiples héritages, c'est la distance la plus longue qui est renvoyée. Cette métrique permet de prédire que:

1. plus une classe est basse dans la hiérarchie de l'arbre d'héritage, plus celle-ci est susceptible d'hériter d'un grand nombre de méthodes, ce qui rend la prédiction de son comportement difficile,
2. plus un arbre est profond, plus cela le rend complexe, car le nombre de classes et de méthodes est plus élevé,
3. plus une classe est profonde dans l'arbre, plus son potentiel de réutilisation de méthodes héritées est grand.

– **NOC (Number Of Children):** NOC renvoie le nombre de classes subordonnées (qui héritent directement dans l'arbre d'héritage) à celle traitée. Chidamber et Kemerer définissent plusieurs uilités à cette métrique:

1. plus NOC est grand, plus il y a de réutilisation, puisque l'héritage est une forme de réutilisation,
2. un grand nombre d'enfants peut aussi signifier un mauvais usage de l'héritage, et une mauvaise abstraction,
3. cette valeur permet aussi d'évaluer l'impact qu'une classe a sur l'architecture du système, plus une classe a d'enfants, plus les méthodes de celle-ci doivent subir de tests.

– **CBO (Coupling Between Objet classes):** Le CBO permet de connaître le nombre de classes qui sont couplées à celle traitée. Nous pouvons considérer que deux classes sont couplées si les méthodes de la première utilisent des méthodes ou des variables d'instance de la deuxième. Cette mesure peut nous amener à faire plusieurs conclusions principales:

1. un couplage excessif entraîne évidemment que la classe sera difficile à réutiliser. Plus une classe est indépendante, plus elle est facile à réutiliser dans une autre application,
2. les couples de classes doivent rester en nombre limité sous peine de voir la difficulté de la maintenance grandir,

3. cette mesure permet également d'évaluer quelle rigueur de test doit être appliquée. Plus le couplage est élevé, plus les tests doivent être pointus.

- **RFC (Response For a Class):** Cette valeur représente le nombre de méthodes susceptibles d'être exécutées en réponse à un message reçu par un objet d'une classe. Les méthodes de cet ensemble sont bien sûr les méthodes de la classe elle-même ajoutées aux méthodes d'autres classes appelées au travers de l'exécution des méthodes de la classe. C'est donc aussi une mesure des potentielles communications inter-classes.

  1. si un grand nombre de méthodes sont susceptibles de répondre à un message reçu, cela complique la phase de test puisque cela demande une plus grande compréhension du comportement de la part du testeur,
  2. plus la valeur de RFC est grand, plus cette classe est complexe.

- **LCOM (Lack of Cohesion in Methods):** LCOM est une valeur représentant le nombre de paires de méthodes qui n'ont aucune similarité diminué du nombre de paires de méthodes qui ont au moins une similitude. Plus le nombre de méthodes similaires est grand, plus la classe fait preuve de cohésion.

  1. la similitude des méthodes d'une classe est un avantage (c'est la preuve d'une bonne cohésion),
  2. une manque de cohésion flagrant peut être solutionné en divisant cette classe en plusieurs sous-classes,
  3. cette métrique permet aussi de repérer des défauts dans l'architecture, le design des classes,
  4. une mauvaise cohésion augmente la complexité et donc le risque d'erreurs durant le développement.

Une partie des métriques de cette suite sera utilisée dans le cadre des outils qui seront présentés par la suite.

2.3.2 Suite de MOOD

Nous pouvons maintenant introduire une autre suite de métriques pour les systèmes orientés objet. Cette suite a été proposée par Fernando Brito e Abreu [eA95] et entre autres évaluée par Rachel Harrison et Steve J. Counsell [HC98], il s'agit de la suite MOOD[1]. Cette suite a été mise au point pour faciliter l'évaluation quantitative du design des logiciels orientés objet. Ces différentes mesures permettent d'exprimer la qualité de la structure interne d'un système, elles sont donc fortement liées à des évaluations de testabilité, de stabilité et de facilité de modification et d'analyse. Nous allons citer quelles sont les

1. MOOD est l'acronyme de "Metrics for Object Oriented Design"

métriques proposées ici qui permettent d'avoir une idée de la qualité du squelette d'un logiciel:

- **MHF (Method Hiding Factor) et AHF (Attribute Hiding Factor):** Ces deux métriques sont proposées comme des mesures d'encapsulation. MHF donne le pourcentage de méthodes cachées d'une classe (c'est à dire qui ne peuvent pas être utilisées par une autre classe). AHF suit le même principe mais en parlant d'attributs cachés à la place de méthodes.
- **MIF (Method Inheritance Factor) et AIF (Attribute Inheritance Factor):** Ces deux mesures permettent de connaître la proportion de méthodes/attributs qui ont été hérités par rapport au total de méthodes/attributs.
- **PF (Polymorphism Factor):** Cette mesure a été proposée comme une estimation du potentiel polymorphique. Le PF est défini comme suit: le nombre de méthodes qui redéfinissent des méthodes héritées, divisé par le nombre maximum de situations de polymorphisme distinctes. C'est donc une mesure indirecte du nombre de *binding* dynamiques dans un système.
- **CF (Coupling Factor):** Le CF est une mesure de couplage inter-classes, à l'exclusion du couplage du au mécanisme d'héritage. Ce facteur représente donc l'importance de la relation entre deux classes et cela pour toutes les paires de classes du système.

Les deux suites qui ont été présentées ici sont très utilisées actuellement dans les recherches concernant les logiciels orientés objet. Il ne fait aucun doute qu'il existe d'autres ensembles de métriques qui ont été proposés, mais dans le cadre de ce travail, ces deux suites ont été exploitées de façon plus importante que d'autres, en particulier celle de Chidamber et Kemerer.

2.4 Visualisation des logiciels

Nous allons ici rapidement faire le tour des outils permettant la visualisation de logiciels puisque cette branche jouera un rôle important dans notre recherche d'indicateur. Maintenant que nous avons vu très brièvement où en étaient les recherches concernant la détection de la qualité de codes orientés objet sur base de méthodes quantitatives et qualitatives, nous allons, en plus de faire un tour de ce qui existe en visualisation logicielle, rapidement présenter les possibilités qu'offre l'outil de visualisation en trois dimensions qui sera utilisé ici.

La visualisation des logiciels est une discipline appréciée et très développée ces dernières décennies. Les différentes techniques utilisent des graphiques, des animations, des schémas, permettant de visualiser le flux de contrôle, d'informations, les dépendances entre modules,... Il existe un grand nombre d'outils permettant chacun de visualiser certains aspect d'un système. Citons par exemple l'outil développé par Michele Lanza et Stéphane Ducasse [LD01] permettant de visualiser la structure interne d'une classe. Nous

pouvons aussi mentionner les outils permettant de visualiser le code réutilisable et donc d'en faciliter sa réutilisation [MAR01].

Il existe également des outils permettant de visualiser la structure générale d'un système, sa répartition en packages et sous-packages par exemple. Une des méthodes les plus connues est la répartition hiérarchique sous forme de *Treemap* (nous verrons dans la partie 3.2.2 (page 39) comment il est possible de représenter une hiérarchie sous forme de *Treemap*). Mais les outils les plus intéressants, dans notre cas, sont ceux permettant de représenter des systèmes entiers, aussi bien leur structure que leurs caractéristiques. Il existe des systèmes se rapprochant de celui que nous exploiterons. Ces systèmes donnent une représentation visuelle d'un système grâce aux caractéristiques de celui-ci. Ces caractéristiques peuvent être, par exemple, des valeurs de métriques [HVW05].

Après avoir parcouru la littérature traitant de la visualisation logicielle, nous nous rendons vite compte que le principal problème de tous les systèmes existants est la représentation complète de grands systèmes. En effet, la multitude d'informations (statique, dynamique, intra et inter-classes, de structure, ...) rend très vite les résultats beaucoup trop compliqués à observer. Une des causes est principalement due à la capacité de l'homme à interpréter de l'information visuelle. Si trop d'informations visuelles sont présentées en même temps, il nous est impossible de discerner tous les détails. C'est d'ailleurs pour cette raison que l'outil de visualisation que nous allons utiliser introduit une nouvelle dimension dans la visualisation des logiciels.

Cet outil de visualisation permet aujourd'hui d'observer en trois dimensions de grands systèmes et de critiquer leurs qualité grâce à la visualisation de données les concernant. A l'heure actuelle, cet outil offre de larges possibilités de détection de la bonne ou mauvaise qualité mais celles-ci sont loin d'être exploitées. Dans la section 3.2 (page 37), nous présenterons en détails ce logiciel.

Chapitre 3

Outils utilisés

Un fois le vocabulaire acquis et avant d'exposer le cadre de la recherche qui a été menée, il est nécessaire de présenter la panoplie d'outils qui étaient à disposition. Comme nous l'avons précisé préalablement, il existe un grand nombre d'outils permettant de calculer des mesures ainsi que d'outils pour visualiser un logiciel. Dans le cadre de cette analyse, nous avons fait des choix arbitraires pour restreindre notre domaine de recherche. Nous utiliserons principalement des mesures orientées objet de la suite proposée par Chidamber et Kemerer [CK94] que nous calculerons grâce à une suite d'outils présentée ci-dessous. Une fois ces calculs de valeurs effectués, nous utiliserons un logiciel de visualisation développé de façon parallèle aux prémisses de cette recherche [LSP05]. C'est une réflexion sur les possibilités qu'offrent ces différents outils qui a initié l'ébauche d'hypothèses et le commencement d'une recherche plus approfondie. Ci-dessous, nous présenterons donc les caractéristiques de chacun des outils et les possibilités que ceux-ci offrent. Dans les parties suivantes, nous exposerons comment ces outils ont été utilisés conjointement pour parvenir à des résultats.

3.1 PTIDEJ

Nous allons tout d'abord présenter certains composants de la suite d'outils PTIDEJ[1] comme le métamodèle créé par rétro-conception, ainsi que la suite de métriques logicielles qui a été non seulement utilisée mais aussi enrichie.

Avant de rentrer dans les détails, il est nécessaire de parler de façon générale de la suite d'outils PTIDEJ. Le projet PTIDEJ est constitué d'un suite d'outils permettant d'évaluer et d'améliorer la qualité des logiciels en faisant la promotion des *design patterns* [GUÉ05]. Cette boîte à outils permet, grâce à son interface utilisateur, de créer le modèle d'un logiciel à partir de son code source, d'identifier des micro-architectures similaires à un *design pattern*, et d'appeler différents générateurs, analyses, et autres outils extérieurs sur le modèle du logiciel. Le projet PTIDEJ est décomposé en plusieurs parties. Elles

1. PTIDEJ est l'acronyme de "Pattern Trace Identification, Detection, and Enhancement in Java"

seront citées brièvement ci-dessous (seules celles qui nous intéressent seront détaillées plus précisément par la suite):

- Un métamodèle PADL[2] pour décrire des logiciels orientés objet.
- Une librairie de *design patterns*.
- Différents parsers pour construire des modèles de logiciels écrits dans des langages différents. Les plus importants sont les parsers pour fichiers C++ et fichiers `.class` Java.
- Une librairie de métriques logicielles, POM[3], permettant de calculer des métriques reconnues sur le modèle du logiciel, telles que les métriques de Chidamber et Kemerer [CK94].
- Une librairie de générateurs et d'analyses pouvant être appliqués sur les modèles du système.
- Un solutionneur de contraintes, PTIDEJ SOLVER, pour identifier des micro-architectures.
- Un analyseur dynamique pour Java, CAFFEINE, basé sur un moteur Prolog et sur l'interface de débogage Java pour définir les relations entre les classes.
- Une librairie graphique, PTIDEJ UI, permettant d'afficher les modèles et *patterns* de programmes, ainsi que des données dynamiques de CAFFEINE.
- Différentes interfaces utilisateurs graphiques pour accéder aux fonctionnalités offertes par la suite PTIDEJ.

Dans les sections suivantes, nous allons expliquer un peu plus en détails les parties de PTIDEJ qui ont été exploitées et enrichies.

3.1.1 PADL

Tout d'abord, décrivons plus profondément le métamodèle PADL sur lequel vont se baser tous les calculs de mesures logicielles qu'effectue POM.

PADL [GUÉ05] est un métamodèle créé à partir de la rétro-conception d'un système, en partant du code Java compilé. La figure 3.1 (page 33) est un diagramme de type diagramme de classes UML qui représente une architecture en couches correspondant au métamodèle PADL. Si nous nous intéressons un peu plus à la couche centrale, on peut remarquer que le métamodèle PADL définit plusieurs constituants (par exemple `IIdiomLevelModel`, `IMethod`) qui sont des interfaces pour décrire les modèles structurels des logiciels et des *patterns* (ainsi qu'un sous-ensemble de leurs comportements).

Comment fonctionne concrètement PADL (en particulier pour du code Java)? PADL crée le métamodèle à partir d'un fichier JAR ou à partir de fichiers `.class` (il faut dans ce cas lui indiquer le répertoire contenant les fichiers). Le code de chaque fichier est parcouru et les différents constituants sont étiquetés selon leur type (interface Java, méthode Java,

2. PADL est l'acronyme de "Pattern and Abstract-level Description Language"
3. POM est l'acronyme de "Primitives,Operators, Metrics"

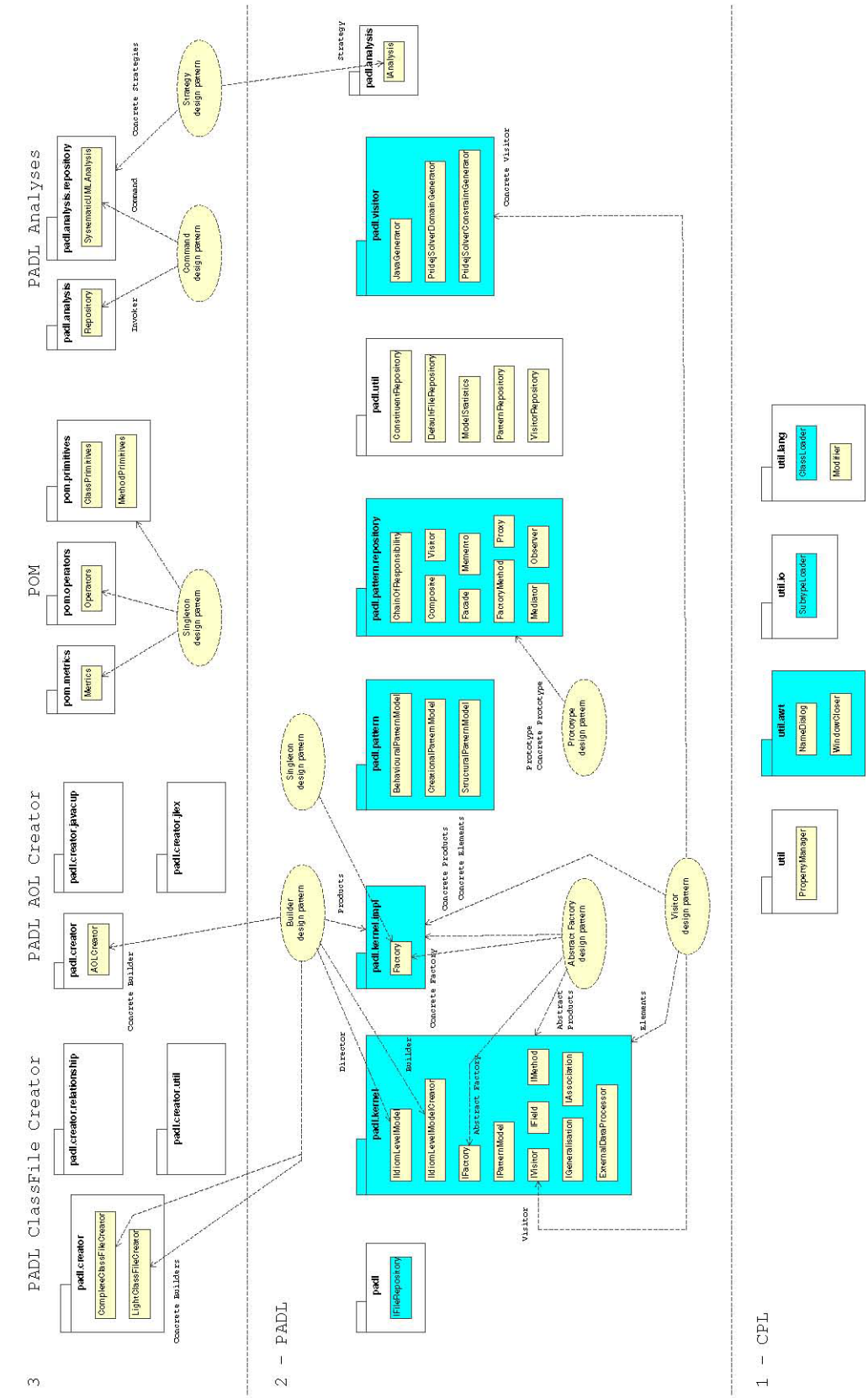

FIG. 3.1 – *Architecture en couches du métamodèle* PADL

...). Une fois tous les constituants étiquetés, nous obtenons un métamodèle complet, représentant en détails les différents composants du système analysé et les relations qui les relient [GUÉ03]. Dans la troisième couche de PADL, on retrouve donc différents projets qui utilisent le modèle façonné dans la couche centrale. On retrouve notamment le projet POM décrit ci-dessous.

3.1.2 POM et les différentes métriques observées

POM [GUÉ05][ZAI04] est une librairie Java permettant de calculer un grand nombre de métriques orientées objet à partir d'un méta-modèle PADL. Les différentes métriques utilisées et observées seront décrites ci-dessous. Ces métriques représentent des propriétés des classes. POM est décomposé en un certain nombre de primitives qui sont définies comme des constituants du métamodèle. Ces primitives sont combinées avec des opérateurs d'ensemble pour implémenter les calculs des métriques. Dans le cadre de cette recherche, plusieurs métriques ont été utilisées et observées. Ces différentes métriques ainsi que leur calcul précis sont définis ci-dessous.

- **CBO (Coupling Between Objects Classes)** Cette métrique représente, pour une classe, le nombre d'autres classes à laquelle elle est couplée. La notion de couplage (entrelacement) est définie comme suit: "Un objet est couplé à un autre si un des deux agit sur l'autre. C'est-à-dire, si une méthode de l'un utilise une méthode, une instance ou une variable de l'autre" [CK94]. Il est évident que dans cette définition, les objets doivent être de classes différentes et que si l'objet X utilise une variable de l'objet Y et inversément, cette double relation ne compte que pour une fois. Le CBO d'une classe est à rapprocher avec la notion de FAN-OUT. Le FAN-OUT d'une classe C est le nombre de classes référencées dans C.

  Un couplage excessif montre des faiblesses dans l'encapsulation de la classe et peut rendre la réutilisation de la classe compliquée. Un couplage élevé indique aussi que plus de fautes pourraient être introduites dans le code à cause de la forte activité inter-classes.

  Grâce au métamodèle qui représente également les liens inter-classes, il est aisé de compter combien de classes sont reliées à la classe observée.
- **DIT (Depth of Inheritance Tree)** La métrique DIT représente la profondeur d'une classe dans l'arbre d'héritage. La profondeur d'un noeud dans un arbre est la longueur du chemin le plus long reliant le noeud à la racine de l'arbre [CK94]. L'outil POM donne une valeur de DIT de minimum 1. Pourquoi 1? Car les classes analysées héritent toujours de la classe `Object` de Java. Nous considérons donc qu'une classe avec une valeur de DIT égale à 1 est la racine d'un arbre d'héritage. POM calcule cette métrique grâce aux relations inter-classes représentées par PADL et est même capable d'indiquer qui sont les parents et enfants de chaque classe.
- **LCOM5 (Lack of Cohesion in Methods, 5^{th} version)** Depuis la version de Chidamber et Kemerer [CK94], beaucoup de critiques ont été apportées à LCOM.

C'est pour cette raison qu'il existe plusieurs versions de cette métrique [BBM96]. Celle qui est implémentée dans POM est la $5^{\text{ème}}$ version proposée par Henderson-Sellers dans [HS96]. Cette version est basée sur le nombre de variables d'instance référencées. Une classe est plus cohésive si un grand nombre de ses variables d'instance sont référencées par ses méthodes. La formule mathématique de LCOM5 est définie ci-dessous:
Considérons un ensemble de méthodes M_i (i=1, ..., m) qui accèdent à un ensemble de variables d'instance A_j (j=1, ..., a). Soit $\mu(A_j)$ le nombre de méthodes qui référencent A_j. Alors

$$LCOM5 = \frac{(1/a)\sum_{1 \leq j \leq a} \mu(A_j) - m}{1 - m}$$

– **WMCC (Weighted Methods per Class, Cyclomatic)** La librairie POM contenait déjà une métrique de "taille", WMC (Weighted Methods per Class). Dans le cadre de ce travail, il était nécessaire de calculer et analyser la complexité cyclomatique [McC76] des différentes méthodes. De plus, comme le système de visualisation représente des propriétés de classe et non pas de méthode, il a été décidé de sommer les différentes complexités cyclomatiques de chaque méthode d'une classe pour rendre une seule valeur par classe. Le constructeur de chaque classe Java est considéré comme une méthode dans le métamodèle. Cette décision amène plusieurs critiques bien fondées. La principale est de dire qu'aucun poids n'est associé à chaque méthode. Le problème est donc qu'une méthode "mal codée", dans le sens où cette méthode est extrêmement complexe, au sens cyclomatique, par rapport au reste des méthodes de la classe, influence fortement la complexité cyclomatique totale de la classe.

Pour définir plus en détail le calcul de WMCC, il est nécessaire de préciser ce qu'est la complexité cyclomatique. Le nombre cyclomatique définit par McCabe a pour but de mesurer la complexité du flux d'instructions d'une méthode ou d'un module. Ce nombre est calculé à partir du graphe de la méthode, du module, qui représente la topologie du flux de contrôle. La formule suivante définit le calcul de ce nombre:

$$CNN = E - N + P$$

où E est le nombre d'arêtes dans le graphe, N le nombre de noeuds et P le nombre de composants connectés.

Ce que le nombre CNN représente concrètement est le nombre de circuits linéairement indépendants au sein de la méthode. Plus CNN est élévé, plus il est complexe de comprendre, corriger et tester la méthode analysée.

Il existe une autre méthode de calculer la complexité cyclomatique d'une méthode, il suffit de compter le nombre de noeuds de décision (`if`, `while`, `for`, `case`, ...) au sein d'une méthode et de l'additionner à 1. Une méthode qui n'aura pas de noeuds

de décision dans son flux d'instruction aura une complexité cyclomatique de 1. Ensuite, il suffit de rajouter 1 à chaque fois qu'un noeud de décision est rencontré. Le résultat de ce calcul donne le même résultat que le précédent, c'est-à-dire qu'il représente le nombre de circuits indépendants au sein d'une méthode.

Cette deuxième méthode a été utilisée pour calculer WMCC avec POM. A la création du métamodèle, le nombre CNN est calculé pour chaque méthode et ce nombre est stocké dans une variable pour chaque méthode du système. Grâce au métamodèle, à partir d'une classe, il suffit donc de prendre la variable CNN de chaque méthode de la classe et de les additionner pour obtenir WMCC.

3.1.3 Modifications apportées à PADL et POM

Le point de départ de cette recherche a été l'envie de pouvoir définir le degré d'"orientation objet" d'un logiciel. Après quelques lectures, notamment [SLPH05], et investigations, il a semblé nécessaire de pouvoir calculer la complexité cyclomatique de McCabe pour essayer de trouver une corrélation entre celle-ci et le degré d'"orientation objet". La métrique WMCC décrite ci-dessus n'était pas présente dans l'outil POM, il a donc été nécessaire de modifier certaines parties de cette librairie ainsi que du code de création du modèle PADL.

Comme expliqué plus tôt, la création du métamodèle PADL se base sur le code compilé Java. Nous avons donc à notre disposition un parser de bytecode Java pour modéliser le système. Les primitives existantes de PADL repéraient un certain nombre d'instructions dans le bytecode Java pour en retirer les informations nécessaires à la rétro-conception du modèle. Pour pouvoir calculer le nombre cyclomatique de McCabe, il était donc nécessaire de repérer les opérateurs de branchement conditionnels dans le bytecode. Les opérateurs correspondants sont: `ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpeq`, `if_acmpne`, `lcmp`, `fcmpl`, `fcmpg`, `dcmpl`, `dcmpg`, `tableswitch` et `lookupswitch` [LY96]. Une fois ces commandes repérées dans le corps de la méthode, il suffit de les compter et d'ajouter 1 au total pour obtenir la complexité cyclomatique d'une méthode. Les modifications apportées à PADL ont donc été la modification du parser de bytecode pour le forcer à réagir à la rencontre de certaines instructions supplémentaires, et l'ajout d'une variable par méthode (méthode au sens d'objet du métamodèle représentant une méthode) qui contient la complexité cyclomatique de celle-ci.

En ce qui concerne POM, il a suffit d'ajouter une métrique dans la librairie et une primitive qui accède à la variable ajoutée dans le métamodèle. Le calcul de WMCC fait simplement appel à la primitive à chaque passage dans la boucle, qui correspond à chaque méthode traitée. Il somme au fur et à mesure ces données pour enfin obtenir le nombre WMCC.

3.2 Outil de visualisation

Cet outil de visualisation a été développé dans le but de pouvoir améliorer le développement et la maintenance des logiciels [LSP05]. Il existe évidement beaucoup d'outils permettant d'analyser des données de façon automatique. Mais la visualisation offre des outils puissants pour développer une meilleure compréhension de la qualité logicielle. La visualisation permet le traitement automatique et la présentation des données d'une manière à ce que l'expert humain puisse identifier rapidement et facilement des *patterns* complexes et des particularités de design significatives. De plus, l'expertise ne peut pas être efficace quand il s'agit de critiquer un système de très grande taille. Par exemple, un diagramme de classes UML contenant des centaines de classes est très difficile à analyser. Dans ce contexte, cet outil propose une approche basée sur la visualisation pour des analyses de systèmes complexes en contournant le problème de la taille grâce à une exploitation importante du système visuel humain. Ce système représente des données qu'on lui donne dans un format XML précis. Dans notre cas, nous lui donnons un fichier généré à partir du métamodèle PADL et des métriques calculées par POM.

Ce système est basé sur quatre aspects importants: la représentation des classes, la représentation des programmes, la navigation et le filtrage des données.

3.2.1 Représentation des classes

Le but étant d'analyser des grands systèmes, la classe est prise comme élément de base. Il est évidement impossible de représenter un logiciel dans sa forme initiale car il n'en a pas. Il a donc été décidé de représenter la classe (élément de base) sous forme d'une boîte géométrique en trois dimensions (un parallélépipède rectangle placé verticalement). Cette forme est utilisée pour des raisons de facilité de perception et aussi d'affichage à l'écran. De plus, comme le but est de représenter des propriétés de classe et de design, il est aisé d'utiliser les propriétés variables qu'offre une boîte 3D pour les faire correspondre à des valeurs telles que des métriques. Ces différentes caractéristiques variables sont limitées au nombre de trois: la taille, la couleur et l'orientation. Ces trois propriétés sont utilisées comme suit:

- **Taille:** Plus une boîte est grande en terme de hauteur, plus la valeur de la métrique qui lui est associée est grande. Typiquement, on associe la complexité cyclomatique (WMCC) à la taille.
- **Couleur:** La couleur des boîtes est en fait une variation entre le bleu et le rouge. Une classe colorée d'un bleu très foncé indique que la métrique associée à la couleur a une valeur très faible. Et plus la couleur vire vers le rouge, plus la valeur de cette métrique augmente. On associe la plupart du temps la métrique CBO à cette variation de couleur.
- **Orientation:** Le terme original pour définir cette caractéristique est "Twist". Il est plus facile de parler d'orientation du parallélépipède rectangle par rapport à l'axe bas-haut et gauche-droite lors de la visualisation. L'outil nous présente les données

de telle façon à ce qu'il soit possible de s'orienter et de naviguer entre les classes, il est donc facile de voir si une classe forme un angle plat ou un angle droit avec l'axe bas-haut. Si une classe forme un angle plat avec l'axe bas-haut, la valeur associée est la valeur la plus basse possible. Ensuite, la classe subit une rotation plus la valeur associée augmente, jusqu'à atteindre le maximum quand la classe forme un angle de 90 degrés avec l'axe bas-haut. On utilise souvent cette orientation pour représenter la métrique LCOM5 ou la métrique DIT (comme on le verra plus tard, il existe une visualisation particulière pour analyser l'utilisation du mécanisme d'héritage).

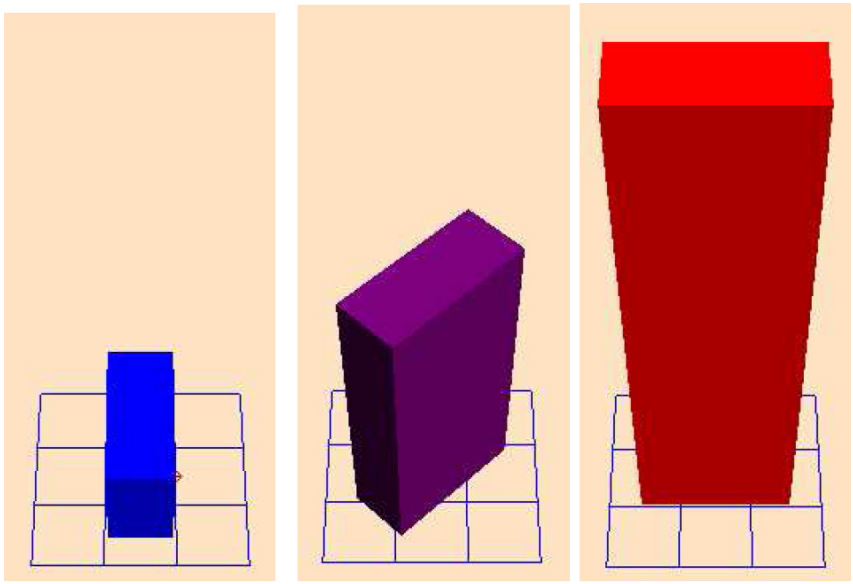

FIG. 3.2 – *Représentation de 3 classes: les valeurs des 3 métriques associées (WMCC, CBO, LCOM5) augmentent de gauche à droite.*

On peut voir sur la figure 3.2 (page 38) comment évoluent ces trois propriétés. Ces associations de métriques aux propriétés ont été choisies car ce sont les plus intuitives. On associe par exemple un couplage élevé à un objet très rouge, car intuitivement, nous associons la couleur rouge au danger, à ce qui est mauvais.

Il est nécessaire de préciser aussi que les classes qui sont des `Interface` Java, sont représentées par l'outil de visualisation sous forme de cylindres. Cette forme a été choisie car les `Interface` Java ont toujours des métriques CBO, DIT, LCOM5 et WMCC insignifiantes. Il n'est donc pas nécessaire d'avoir plusieurs propriétés modifiables comme dans le cas des boîtes en trois dimensions utilisées pour représenter les autres classes Java. Précisons encore que dans notre utilisation de cet outil, nous ferons en sorte que ces `Interface` Java ne soient pas prises en compte et n'apparaissent donc pas à l'écran. Les `Interface` Java, quand elles sont représentées à l'écran, peuvent fausser la visualisation et la critique car elles n'introduisent qu'un billet. Vu que nous nous plaçons à un niveau élevé pour observer le système, un grand nombre de classes `Interface` dans un package

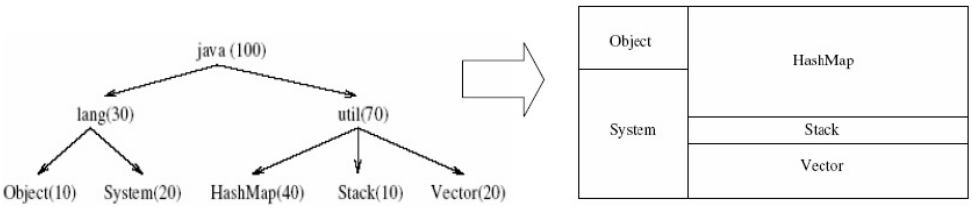

FIG. 3.3 – *Illustration du fonctionnement de la technique Treemap*

fausse la visualisation en introduisant beaucoup de couleur bleue (vu que les `Interface` Java ont un CBO très faible, voire nul) dans la zone observée. Ce qui peut biaiser notre critique vis-à-vis du couplage général du système.

3.2.2 Représentation des programmes

Maintenant que nous avons décrit comment les objets de bases sont représentés, il faut expliquer comment ces objets sont disposés et organisés entre eux. Ce système propose deux types de disposition: *Treemap* et *Sunburst*.

Treemap

La disposition *Treemap* originale commence par représenter un rectangle qui correspond à la racine d'une hiérarchie. Ce rectangle est divisé en un nombre de tranches verticales égal au nombre des enfants de la racine. Ces nouveaux rectangles sont à leur tour coupés, mais cette fois horizontalement, en d'autres plus petits rectangles internes. Ce processus de division continue jusqu'à ce que tous les noeuds de la hiérarchie soient représentés. Pour organiser les classes dans cet environnement 3D selon l'algorithme *Treemap*, nous prenons comme racine de la hiérarchie, l'emplacement racine du logiciel. Ensuite ce rectangle racine est divisé verticalement selon le nombre de packages que contient l'emplacement racine du programme. Ensuite, les rectangles packages sont chacun divisés selon leurs sous-packages, et ainsi de suite. La figure 3.3 illustre cette technique. La première partie de la figure 3.5 (page 41) montre une représentation de cette technique dans l'environnement 3D.

Sunburst

La disposition *Sunburst* fournit une distribution circulaire d'une hiérarchie. Un cercle central représente la racine du logiciel. Ensuite ce cercle est divisé en un nombre de parties correspondant au nombre de packages que contient le répertoire de base. Les packages de même niveau sont séparés par des droites qui coupent les arcs de cercle en portions, tandis que les packages de niveaux différents sont séparés par des arcs de cercle, de façon à ce que, plus un package est profond dans l'arbre de la hiérarchie, plus il est éloigné du centre. La figure 3.4 (page 40) montre comment une hiérarchie est représentée en *Treemap* et en

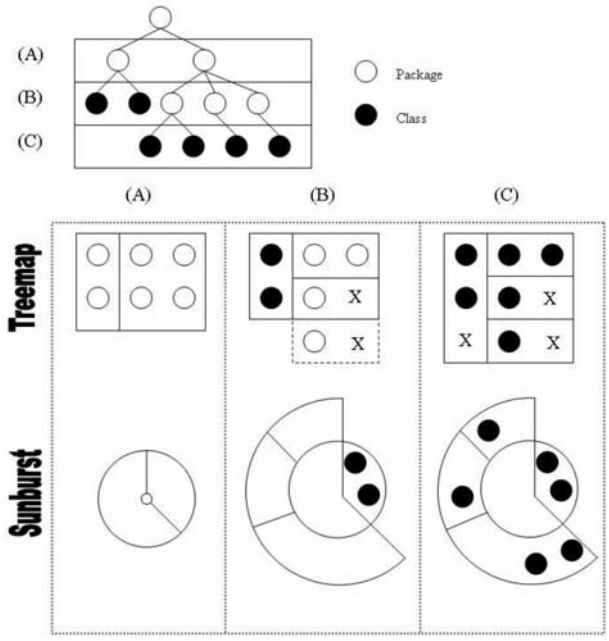

FIG. 3.4 – *Illustration de la conversion d'une hiérarchie grâce aux techniques Treemap et Sunburst*

Sunburst.

La disposition *Sunburst* apporte un avantage intéressant pour la visualisation de l'utilisation du mécanisme d'héritage. En effet, il est aisé de modifier les données utilisées par l'outil de visualisation pour qu'il représente le système sous forme d'un graphique de type "secteurs". Dans le fichier XML que le système demande en entrée, il est possible de repérer les différentes classes et leurs métriques associées. Un petit logiciel existe donc pour modifier ce fichier pour que chaque classe soit placée dans la même portion que les autres classes qui ont le même DIT. On obtient donc un cercle divisé en portions représentant les différents niveaux de l'arbre d'héritage, et donc l'importance du nombre de classes qui se situent aux différents "étages" de la hiérarchie. La figure 3.6 (page 42) donne un exemple de cette possibilité. Il faut préciser qu'un programme est fait de plusieurs arbres d'héritage et que cette représentation ne nous permet pas de faire la différence entre chacun.

3.2.3 Navigation

Un des points forts de ce système, est qu'il est possible de naviguer au coeur de la représentation 3D. On peut faire la comparaison avec la gestion de la caméra dans un jeu vidéo. Il est possible de faire des rotations, de faire des zooms, et de se déplacer dans toutes les directions au sein de cet espace 3D. Ces possibilités permettent d'observer des phénomènes qui pourraient être cachés par d'autres, comme une classe cachée derrière une autre classe fortement complexe, donc de grande taille.

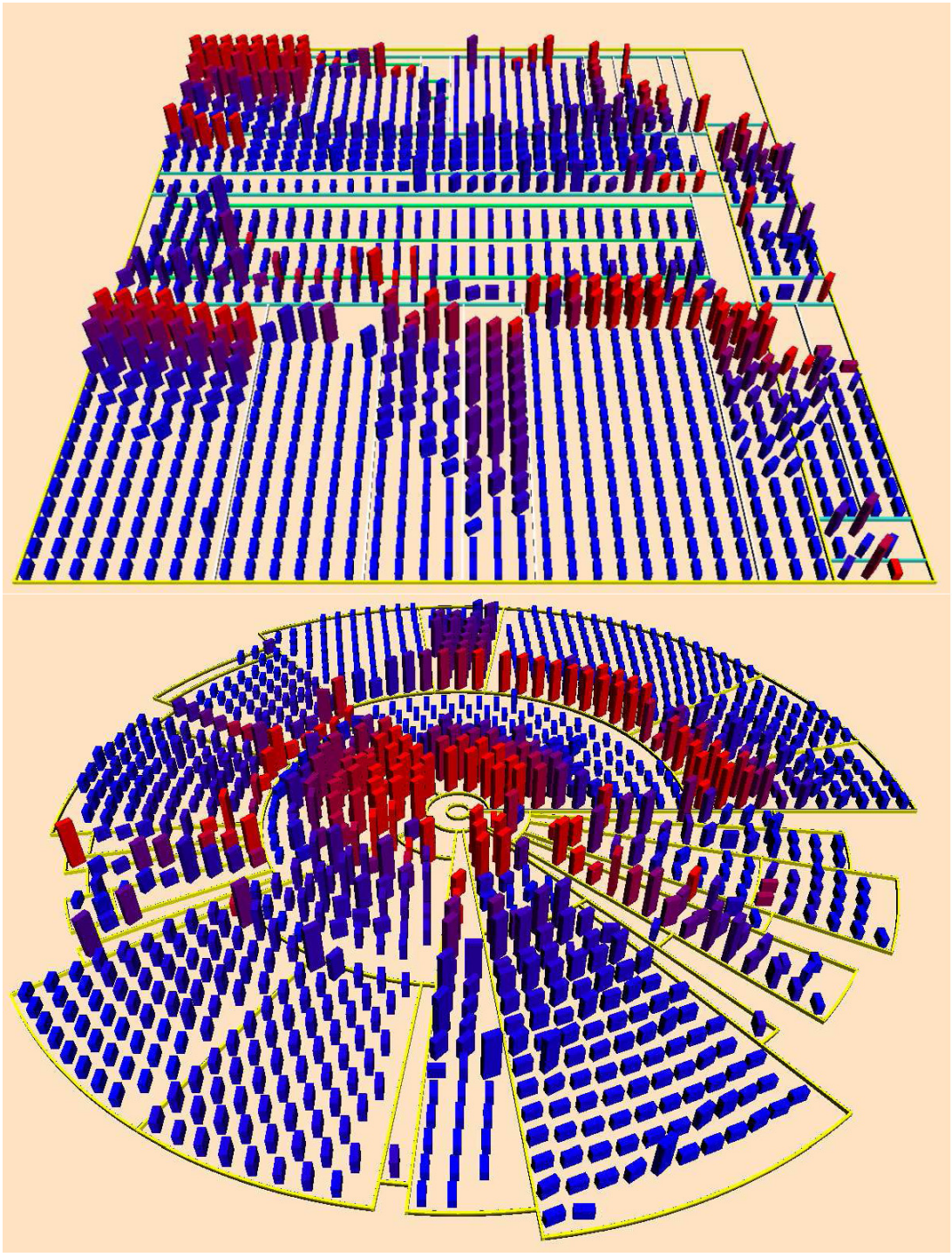

FIG. 3.5 – *(Haut) Technique Treemap et (bas) technique Sunburst représentant toutes les deux le logiciel Java PCGEN, un outil de génération de personnages pour jeux de rôle.*

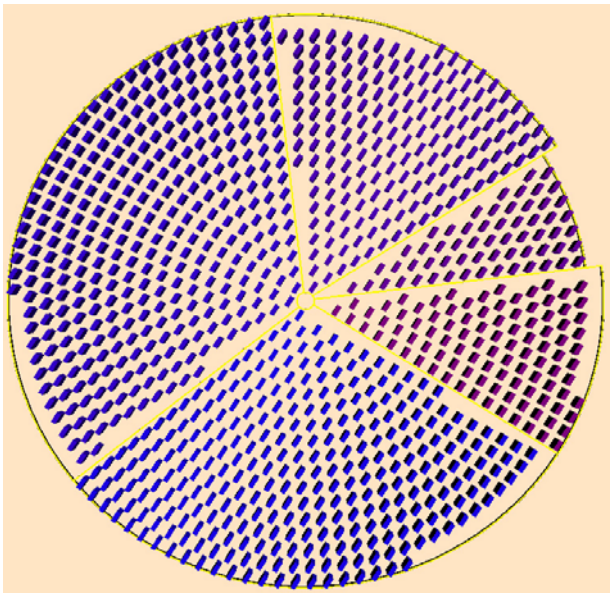

FIG. 3.6 – *Représentation du mécanisme d'héritage.*

3.2.4 Filtrage de données

Pour certaines analyses, il est important de pouvoir se focaliser sur un sous-ensemble de données tout en gardant une vue globale du système. L'outil de visualisation nous propose différents types de filtres pour mettre en évidence certaines données ou pour réduire leur impact visuel en retirant leur coloris. Deux filtres principaux sont à notre disposition.

Le premier est un filtre basé sur les métriques. Il est par exemple possible de mettre en évidence les classes qui ont des valeurs extrêmes pour certaines métriques. Ce filtre donne la couleur rouge aux classes qui ont des valeurs extrêmes et donne la couleur verte ou jaune aux autres. Les valeurs extrêmes sont détectées grâce à des techniques classiques de statistique telles que "la boîte à moustache" (*"plot box diagram"*). Un exemple de l'application de ce filtre est visible sur la figure 3.7 (page 43).

Le deuxième filtre proposé permet d'extraire des relations UML inter-classes. Pour une classe particulière, l'expert peut demander à mettre en évidence les classes qui sont liées à celle qu'il analyse par un lien particulier (classes parentes, filles, couplées, ...). Les classes liées restent colorées, tandis que toutes les autres perdent leur coloris. La figure 3.8 (page 43) illustre l'utilisation de ce filtre.

Cette section conclu la présentation des différents outils.

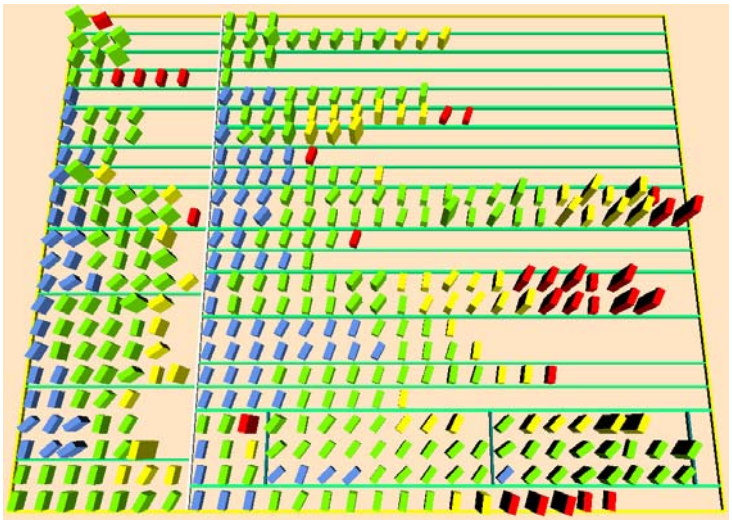

FIG. 3.7 – *Représentation du filtre "Plot Box".*

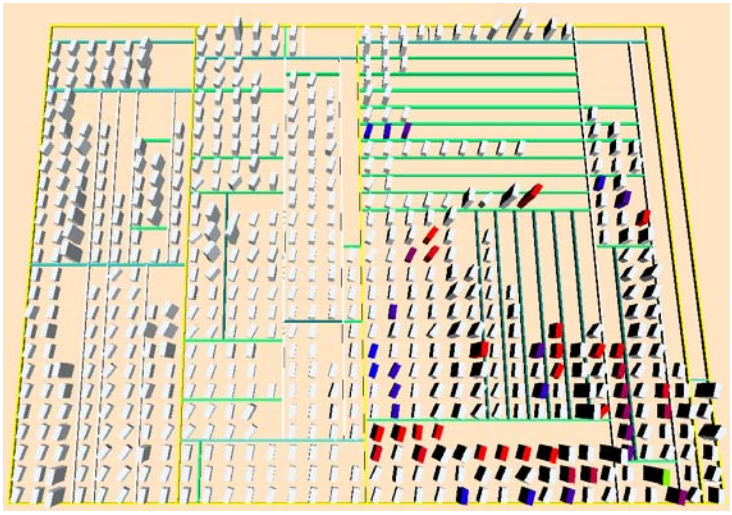

FIG. 3.8 – *Représentation du filtre "UML". La classe colorée en vert en bas à droite de l'image est la classe analysée. Toutes les autres classes colorées sont celles qui ont des relations de couplage avec la classe analysée.*

Deuxième partie

Méthodologie, développement et résultats

Chapitre 1

Etapes de la recherche

1.1 Cadre et orientation de la recherche

Dans le cadre d'un stage et d'un mémoire de fin d'étude, le professeur Naji Habra des Facultés Universitaires Notre-Dame de la Paix (Namur), et le professeur Houari A. Sahraoui de l'Université de Montréal ont proposé de mener une réflexion sur la qualité des logiciels orientés objet. Cette réflexion tend à se placer dans un courant de littérature abordant la critique de la qualité des logiciels orientés objet, notamment dans un souci d'amélioration de la maintenabilité des logiciels. Il est évident que la notion de maintenabilité est une caractéristique principale de la qualité orientée objet. Le paradigme Orienté Objet a la réputation de faciliter la maintenance vu son aisance de compréhension et de réutilisation. Il est donc primordial de relier la qualité orientée objet, dans le sens de bonne utilisation des principes objets, à la notion de maintenabilité.

Cette recherche a donc pour but le développement d'un ou plusieurs indicateurs orientés objet permettant de juger de certaines caractéristiques de la qualité orientée objet d'un système, telle que la maintenabilité. En d'autres mots, cette réflexion doit mener à l'aboutissement du développement d'une panoplie d'outils permettant de dire si un système est un "bon" système orienté objet, ou non. La notion de "bon" ou "mauvais" code orienté objet n'est pas définie actuellement dans la littérature. Nous interpréterons cette notion dans le sens de bonne ou mauvaise utilisation des principes orientés objet. Nous entendons par "principes orientés objet", les mécanismes tels que l'encapsulation et l'héritage. Il faut donc différencier la "bonne" utilisation des principes orientés objet et le "degré" d'utilisation de ces principes. Cette idée s'inscrit dans le courant de la recherche suivie au laboratoire de génie logiciel de l'Université de Montréal, où sont menées, par exemple, des recherches sur la détection d'*anti-patterns* et autres défauts de conception dans les codes orientés objet.

1.2 Historique de la démarche

1.2.1 Prémisses

L'idée principale de la recherche qui devait être menée est celle du développement d'indicateurs de certaines qualités orientées objet par une expérimentation rendue plus facile grâce à un outil de visualisation. En effet, dans le laboratoire de génie logiciel de l'Université de Montréal, des recherches sont menées à propos de la visualisation des logiciels. Un outil permettant de visualiser de grands systèmes orientés objet a été développé ces dernières années. Comme expliqué dans la section 3.2 (page 37), cet outil permet de représenter les classes d'un système en modifiant l'apparence des objets trois dimensions représentant ces classes selon certaines valeurs, ici décidées par le calcul préalable de métriques orientées objet sur ces classes. L'idée principale retenue était donc d'essayer de développer des indicateurs sur base de ce système permettant de "voir" le bon ou mauvais code orienté objet. Actuellement, ce logiciel est utilisé pour détecter des défauts de conception, ou des mauvaises habitudes de codage, telles que les *anti-patterns*.

Les outils que nous utilisons ici proviennent pour la plupart du laboratoire de génie logiciel de l'Université de Montréal. Dans l'ordre, le système de rétro-ingénierie PADL a été développé. Ensuite, la suite de métriques orientées objet POM est venue s'ajouter à celui-ci. Aujourd'hui, tous ces outils forment une suite complète qui ne cesse d'évoluer, cette boîte à outils est l'ensemble PTIDEJ que nous avons présenté précédemment. Le système de visualisation en 3D de logiciels a été développé en parallèle. Nous l'utilisons avec des données produites par les métriques de POM mais il est évident que ce système peut s'utiliser avec d'autres données et qu'il ne rentre donc pas dans la suite PTIDEJ. C'est dans ce cadre de développement que nous avons combiné tous ces outils pour tenter de produire des indicateurs orientés objet efficaces.

1.2.2 Idée et objectif secondaire

Comme nous le verrons plus tard, pour pouvoir émettre des hypothèses intéressantes sur ce qui pouvait constituer un indicateur de bon ou de mauvais code orienté objet, il a été nécessaire de faire un grand nombre d'observations avec l'outil de visualisation. A force d'observer sous toutes les formes un grand nombre de logiciels, une seconde idée a surgi. Ces observations nous ont conduits à une deuxième hypothèse indépendante de celle concernant les indicateurs visuels de degré d'orientation objet. Nous avons souvent constaté que lorsqu'on observait des caractéristiques telles qu'un couplage excessif dans le système, en parallèle avec une mauvaise cohésion et une utilisation faible ou incohérente du mécanisme d'héritage (ces trois défauts témoignent d'une mauvaise architecture orientée objet), nous pouvions également remarquer un grand nombre de classes présentant une complexité cyclomatique très élevée (selon les valeurs représentées de la métrique WMCC mise au point dans le cadre de ce travail). Ces observations multiples nous ont conduits à l'idée qu'il était possible qu'il y ait une corrélation entre le couplage, la cohésion et le mécanisme d'héritage observés ensemble, et la complexité cyclomatique, ces quatre pro-

priétés observées au niveau du système tout entier. Cette hypothèse un peu audacieuse et utopique signifierait donc que seule l'observation de la complexité cyclomatique au niveau du système tout entier nous permettrait de juger de la qualité orientée objet du système.

Cette idée nous est non seulement venue suite à un grand nombre d'observations mais aussi parce que des recherches similaires avaient déjà été menées. Par exemple l'idée, tirée de l'article [SLPH05], était de trouver une relation entre la complexité cyclomatique de MacCabe et le degré d'orientation objet. Intuitivement, nous tendons à dire que plus un programme, par exemple un logiciel codé en Java, respecte les bons principes orientés objet pour limiter le couplage inter-classes, pour optimiser la cohésion et le mécanisme d'héritage,... moins celui-ci sera complexe au sens de la complexité cyclomatique de MacCabe. L'hypothèse qui a donc été retenue de cette littérature était que la complexité cyclomatique, observée d'une façon particulière, pouvait être à elle-seule un indicateur de bon ou mauvais code orienté objet. Pour tester cette hypothèse, il sera nécessaire de mettre en relation des métriques orientées objet (DIT, CBO, LCOM5 dans ce cas-ci) avec la complexité cyclomatique et essayer de trouver une corrélation entre ces métriques orientées objet mises ensemble et la complexité cyclomatique.

Une fois ces deux objectifs réalisés, il serait évidemment intéressant et valorisant de pouvoir présenter les résultats obtenus à une communauté plus large d'experts pour voir si ces indicateurs sont validés par la majorité ou non. Nous parlerons donc ici d'un second type de validation. La première validation que nous effectuerons pour le second objectif concernant la complexité cyclomatique sera une validation mathématique sur base de recherche de corrélation pour valider nos observations (complexité élevée quand on observe un mauvais couplage général, une mauvaise cohésion générale et une utilisation faible ou incohérente de mécanisme d'héritage). Nous appellerons le second type de validation, consistant à confronter nos résultats et nos méthodes à d'autres techniques ou à d'autres résultats obtenus avec d'autres données, une validation empirique se basant sur le jugement d'experts plutôt que sur des résultats mathématiques.

Chapitre 2

Méthodologie

Dans ce chapitre, nous décrirons la méthodologie qui sera suivie pour développer de manière rigoureuse et objective les indicateurs orientés objet. Le but étant de pouvoir faire valider nos résultats par une communauté plus large d'experts, une méthodologie de style scientifique sera utilisée. Nous allons donc confronter nos hypothèses à des observations empiriques réalisées sur un échantillon de logiciels orientés objet.

Les étapes de développement décrites ici seront reprises par la suite dans le chapitre suivant où elles seront développées dans le même ordre.

2.1 Hypothèses

Pour commencer, il est nécessaire de définir clairement quelles sont les hypothèses que nous retiendrons tout au long du développement de notre théorie.

1. Notre première hypothèse est: *"Il est possible de juger de la qualité d'un système orienté objet par simple analyse visuelle, sans devoir se plonger dans l'analyse de valeurs multiples"*.
2. Notre deuxième hypothèse, découlant de la recherche pratiquée pour valider la première, est de dire: *"La complexité cyclomatique, analysée au niveau du système entier, peut, à elle seule, constituer une indicateur suffisamment fiable de bonne utilisation des principes orientés objet"*.

Comme nous l'avons déjà dit, c'est suite à la découverte et à l'exploitation des possibilités qu'offrait le nouveau système de visualisation de logiciels (permettant d'observer un grand système en un seul regard) que nous avons remarqué qu'une simple observation visuelle suffisait peut-être pour critiquer la qualité d'un programme orienté objet. Il est évident que cette hypothèse est audacieuse car elle nous épargne de devoir analyser des valeurs de métrique par centaines quand nous parlons de larges systèmes. Nous verrons comment nous avons tenté de valider cette hypothèse et à quels résultats cette recherche nous a menés.

La deuxième hypothèse est encore plus ambitieuse. Notre première idée va nous mener à une façon d'observer un système visuellement, c'est-à-dire à la définition d'indicateurs constitués de *patterns visuels* observables. Par contre, la deuxième hypothèse tend à affirmer que même cette simplification d'observation ne serait pas encore la méthode la plus rapide pour juger de la qualité orientée objet d'un système. Nous montrerons plus tard comment nous sommes parvenus à cette idée et comment nos résultats valident en partie cette affirmation, même si beaucoup reste à faire.

2.2 Problématique

Ces deux idées que nous avions envie de développer amenaient chacune leur lot de problèmes. Tout d'abord, si nous voulions pratiquer des observations en grand nombre, il était nécessaire de constituer un échantillon représentatif de logiciels orientés objet.

Ensuite, avant de pouvoir créer un indicateur de qualité de code orienté objet, n'est-il pas nécessaire de passer par une étape intermédiaire? Car comme expliqué précédemment, pour définir si un code est un bon code orienté objet, il faut observer plusieurs de ses caractéristiques comme le couplage et la cohésion par exemple. Or, à l'heure actuelle, il n'existe pas de guide permettant de savoir comment interpréter ce que l'on voit avec l'outil de visualisation que nous avons exploité. Cet outil permet de cerner certains défauts d'un programme, certaines propriétés du design d'un système, mais il faut savoir les repérer et ensuite les interpréter.

Comme expliqué dans la section "Vocabulaire" (2.1 page 19), nous observerons des propriétés de classe et du design. Une fois la technique d'observation acquise, l'expert conclura en choisissant des types de mauvaise utilisation de l'Orienté Objet qui caractérisent le système observé. Ces types de défauts formeront eux-mêmes une panoplie d'indicateurs qui mettront en évidence des problèmes trouvés dans un système (trop de couplage, mauvaise cohésion, ...), ou, au contraire, qui mettront en évidence les qualités d'un logiciel (couplage faible et bien réparti, arbre d'héritage bien équilibré, ...). Cette étape pose un autre problème. A partir de quand, par exemple, le couplage est-il excessif dans un package? Les types de mauvaise utilisation proposés devront donc être expliqués en détails, avec des exemples illustrant les différents niveaux de "gravité".

Enfin, une fois tous ces indicateurs utilisés, l'observateur devra les prendre tous ensemble pour conclure si oui ou non, le système observé est un bon programme orienté objet. Cette dernière étape soulève plusieurs questions. Tout d'abord, comment l'expert peut-il conclure sur base de plusieurs indicateurs indépendants? Quelle pondération doit-il donner à chacun? Ensuite, qu'est-ce qu'un bon logiciel orienté-objet? Pour répondre à ces questions, il sera nécessaire de définir une technique précise d'observation et de critique pour guider l'expert. Enfin, des exemples de comparaison devront être disponibles pour se référer à des systèmes déclarés comme bons exemples de systèmes orientés objet ou comme mauvais exemples de systèmes orientés objet.

2.3 Étapes de la démarche empirique

Une fois les différents problèmes que nous avons rencontrés et dont la résolution semblait nécessaire à l'avancement de la recherche exposés, nous pouvons présenter les étapes du développement suivies pour parvenir aux résultats qui seront dévoilés plus tard.

2.3.1 Fixation des objectifs

Il est évident qu'il faut d'abord fixer des objectifs clairs formant un plan à suivre. En terme d'objectifs, la démarche qui sera suivie ici vise à fournir une méthode et des outils qui permettront de juger de la qualité d'un système orienté objet. Plus concrètement, nous visons les objectifs suivants:

- La première étape sera bien évidemment de réunir un grand nombre de systèmes orientés objet pour disposer d'un échantillon représentatif. Il faudra ensuite exploiter cet échantillon pour repérer certains phénomènes facilement observables. Nous ferons de ces phénomènes des types de mauvaise utilisation de l'Orienté Objet (voir section 2.1 page 19) en les décrivant précisément et en donnant des exemples de comparaison.
- La seconde étape sera de créer un "guide de visualisation" pour donner à l'expert observateur les outils nécessaires à la détection rapide et précise de ces types de mauvaise utilisation. Ce guide n'aura nullement le but d'influencer l'analyse des systèmes mais bien de jouer le rôle d'une *checklist* permettant de faire une critique objective des systèmes observés. Ce guide permettra dans un premier temps la détection des propriétés de classe, de design et des types de mauvaise utilisation de l'Orienté Objet plus généraux. Il donnera ensuite la méthode à suivre pour juger de la "gravité" de ces défauts d'utilisation observés, autrement dit, pour juger de l'importance à donner à une mauvaise utilisation observée dans le jugement final.
- Une fois ce guide de visualisation et de critique réalisé, nous posséderons une panoplie d'indicateurs de qualité orientés objet (indicateurs de couplage, de cohésion,... d'un système entier). L'objectif à long terme sera de faire valider ces indicateurs par des experts tiers et de les comparer à d'autres techniques de critiques de qualité de logiciels orientés objet.
- Le dernier objectif que ce mémoire vise à accomplir est celui de la validation de la complexité cyclomatique comme indicateur de "bonne" ou "mauvaise" utilisation de l'Orienté Objet (observée seule et au niveau d'un système entier). L'objectif sera donc de trouver une corrélation entre les critiques sur le couplage général et la cohésion générale du système, sur l'utilisation du mécanisme d'héritage au sein du logiciel et les critiques apportées à la complexité cyclomatique générale du programme observé. Toutes ces critiques doivent être réalisées avec le guide précédemment cité.

2.3.2 Création d'un échantillon d'étude

La première étape a évidemment été de créer un échantillon représentatif de systèmes orientés objet. Pour s'assurer d'une certaine objectivité concernant le choix des systèmes, nous avons pratiqué par hasard. Les seules conditions qui devaient êtres respectées étaient le fait que les logiciels devaient être libres de droit (pour la facilité d'accès aux sources dont nous avions besoin) et le fait que les logiciels devaient être assez grands, en terme de nombre de classes. Cette deuxième condition nous assurait des programmes avec une architecture complexe et non pas des systèmes d'une cinquantaine de classes dont l'observation en trois dimensions n'apportait aucun intérêt. La section 3.1 (page 57) détaille un peu plus cette étape.

2.3.3 Calculs des données

Une fois notre échantillon de logiciels établi, il nous fallait extraire les données que nous devions observer. Comme vu dans la section décrivant l'outil de visualisation (3.2 37), l'affichage en trois dimensions du système à observer consiste non seulement en une représentation dans l'espace tri-dimensionnel de sa répartition en packages mais aussi de certaines de ses caractéristiques au choix. Dans notre cas, il est évident que nous allons profiter de cette possibilité pour que les propriétés des objets trois dimensions nous permettent de visualiser des valeurs de métriques. Pour cela, il est bien évident que nous devons calculer toutes ces valeurs de métriques au préalable. Les outils utilisés pour cette étape sont les systèmes PADL et POM. Cette étape est également détaillée dans la section 3.1 (page 57).

2.3.4 Création des indicateurs

Une fois les systèmes récoltés et leurs valeurs obtenues (étape qui a demandé un certain temps), nous pouvions nous lancer dans l'observation de tous ces logiciels. Une première observation de "reconnaissance" a été effectuée permettant de rapidement repérer les systèmes "extrêmes" constituant de bons ou de mauvais exemples de certaines caractéristiques orientées objet. Une fois cette reconnaissance effectuée et le logiciel de visualisation maîtrisé, il ne restait plus qu'à élaborer un "guide" de visualisation documentant la technique de repérage des propriétés de classes et de design intéressantes, c'est-à-dire qui influencent en partie notre jugement final de la qualité, et sur lesquels il faut s'attarder. C'est grâce au repérage de ces *patterns* que nous pourrons définir ensuite des types de mauvaise utilisation de l'Orienté Objet qui nous permettront d'indiquer la qualité orientée objet d'un système. Dans la section 3.1 (page 57), nous verrons de quelle manière nous avons élaboré ce "guide" et les différents indicateurs.

2.3.5 Validation des indicateurs

Quand nous aurons élaboré nos nouveaux indicateurs à utiliser avec le système de visualisation en trois dimensions, il serait intéressant de pouvoir présenter ces résultats à

une communauté d'abord assez restreinte, et ensuite plus large en fonction de la réaction reçue.

Cette recherche n'ayant pas été faite seul, plusieurs personnes se sont déjà penchées sur cette manière de critiquer un logiciel et des résultats assez satisfaisants ont été obtenus grâce à cette technique. Nous expliquerons à qui et comment la technique a été présentée et quelle a été la réaction. Ensuite, nous verrons comment il serait possible de faire valider ces indicateurs par une plus grande communauté.

2.3.6 Réflexion à propos de la complexité cyclomatique

Suite à l'élaboration du "guide" de visualisation permettant facilement de repérer certaines propriétés de classes et de design d'un système orienté objet, et ainsi de mener vers une conclusion de bon ou de mauvais Orienté Objet, nous avons entrepris une seconde observation de l'échantillon. Cette fois, nous avons observé la complexité cyclomatique en plus des autres caractéristiques que nous observions déjà (couplage, cohésion et mécanisme d'héritage). C'est suite à ces observations que nous avons remarqué que certains types de défaut d'utilisation que nous observions (système avec un couplage général élevé, mauvaise cohésion générale du système, ...) était souvent accompagnée d'une haute complexité cyclomatique générale du système. Suite à cette intuition, nous avons décidé de pousser la recherche plus loin.

La méthode qui a été utilisée pour tenter de conforter notre intuition a été d'utiliser le "guide" de visualisation que nous avions créé pour donner une évaluation chiffrée à différentes caractéristiques d'un système. Les caractéristiques qui ont été choisies sont celles qui sont les plus représentatives d'un système orienté objet, autrement dit: le couplage inter-classes général du logiciel, sa cohésion générale, l'utilisation du mécanisme d'héritage et la complexité cyclomatique observée de façon globale. Quand nous parlons d'observation "générale" ou "globale", c'est pour indiquer qu'il s'agit d'une observation sur base de l'outil de visualisation en trois dimensions qui affiche une représentation du système entier. Nous jugeons donc l'impression générale que nous donne le système selon plusieurs critères.

Les critiques se baseront également sur des systèmes repères de notre échantillon que nous considérons comme de bons ou de mauvais exemples en terme d'attributs de qualité orientée objet observables selon notre méthode. Sur cette base, nous allons donner un certain nombre de points (selon une méthode d'attribution de scores) pour le couplage, la cohésion, l'héritage et la complexité de chaque système observé. Ensuite, nous allons essayer de trouver certaines corrélations entre toutes ces données, pour voir si, en effet, la complexité cyclomatique peut-être un indicateur de qualité orientée objet. Nous verrons dans le chapitre 3.5 (page 73) plus en détails comment nous avons procédé et surtout quels résultats nous avons obtenus.

2.3.7 Validation de la complexité cyclomatique

L'intuition que la complexité cyclomatique peut être un indicateur d'un certain "degré d'orientation objet" est une idée à la mode. Nous le voyons encore avec certains récents articles [SLPH05]. Nous pensons donc nous inscrire dans ce mouvement et donc ne pas être totalement ignorés vu que nous apportons notre pierre à un édifice qui est déjà en construction. Nos résultats, comme les précédents, devraient donc, pour atteindre une certaine reconnaissance, être présentés à une plus large communauté que les personnes ayant apporté leur contribution à cette recherche. Ce qui pourrait mener un jour au fait que la complexité cyclomatique soit reconnue comme indicateur de bon ou de mauvais Orienté Objet.

Nous montrerons plus tard comment nous avons obtenu des corrélations prometteuses qui constituent une première validation mathématique et pourquoi nous pensons que celles-ci ne sont pas dues au hasard et nous donnerons des exemples représentatifs. Nous donnerons également la méthode qui sera nécessaire pour faire valider à plus grande échelle et de façon empirique ces résultats.

2.4 Objectifs réalisés

Nous allons décrire ici les différents objectifs qui ont été réalisés dans le cadre de la rédaction de ce mémoire. Les derniers objectifs qui consistent essentiellement en la validation empirique des résultats obtenus restent à réaliser dans une recherche future.

- Un échantillon hétérogène et le plus représentatif possible a été constitué avec les moyens qui nous étaient disponibles, c'est-à-dire que les 50 systèmes de cet échantillon forment un ensemble de petite taille mais qui était largement suffisant pour une recherche et une observation entièrement non automatique. Les données nécessaires (métriques orientées objet) ont été extraites pour chaque système de cet ensemble.
- Cet échantillon a été analysé en détails. De ces analyses visuelles, nous avons pu définir et mettre sur pied un "guide" de visualisation. Des types de mauvaise utilisation des principes orientés objet ont été définis pour permettre à un expert d'utiliser ces outils comme indicateur de qualité orientée objet d'un système.
- Une seconde analyse a été pratiquée pour nous permettre d'essayer de valider notre seconde hypothèse concernant la complexité cyclomatique. Nous avons, grâce à cette seconde analyse, pu établir un tableau de scores attribués aux caractéristiques orientées objet que sont le couplage, la cohésion et l'héritage ainsi qu'à la complexité cyclomatique. Sur base de ce tableau, des recherches de corrélation ont été pratiquées et un premier résultat intéressant a été obtenu. Les données ont été collectées et présentées de nombreuses fois de façons différentes. Les outils et les données sont donc à disposition pour permettre de futures recherches.

Chapitre 3

Démarche empirique

3.1 Création de l'échantillon et extraction de données

3.1.1 Collecte des systèmes

Comme expliqué dans la méthodologie, nous allons tout d'abord rassembler un grand nombre de systèmes orientés objet pour constituer un échantillon observable. Dans un souci de facilité et d'accessibilité, les différents systèmes proviennent de la communauté *Open Source* (via le portail *SouceForge.net*) et sont tous des systèmes codés en Java. Le premier échantillon de programmes qui a été constitué rassemblait environ 100 systèmes différents. Cet échantillon a été réduit à un nombre de 50 systèmes. Les 50 systèmes les plus larges en terme de nombre de classes et les plus hétérogènes en terme de fonctionnalités ont été choisis. Les plus larges ont été sélectionnés simplement parce qu'en terme d'analyse visuelle de haut niveau, il est plus intéressant d'avoir un grand nombre de classes et de packages à observer. Les petits systèmes ne permettent pas de faire des analyses critiques de haut niveau car il est difficile d'observer les caractéristiques d'un grand nombre de classes à la fois. Une description de cet échantillon et des systèmes qui le constituent est disponible en annexe.

3.1.2 Extraction des données

La deuxième étape de la création de l'échantillon a été de procéder au calcul de toutes les mesures dont nous avions besoin pour observer les systèmes. Grâce aux outils de la suite PIDEJ, nous avons procédé à la rétro-ingénierie des 50 systèmes de notre échantillon pour en extraire les valeurs de plusieurs métriques orientées objet, telles que CBO, LCOM5 ou encore DIT. Un métamodèle de chaque système a donc été créé grâce à l'outil PADL pour qu'ensuite nous puissions utiliser l'outil POM pour calculer toutes les données nécessaires. Nous avons donc obtenu un fichier XML par système contenant toutes les données nécessaires à leur affichage en trois dimensions.

3.1.3 Première observation

Pour pouvoir dégager des hypothèses, une première observation des systèmes a été pratiquée. Celle-ci a permis de pouvoir se familiariser avec le système et surtout de pouvoir extraire les caractéristiques, les propriétés de design, facilement observables.

L'hypothèse qui est formulée après cette première observation, est *"Peut-on cerner les caractéristiques de design des systèmes orientés objet par simple observation visuelle, sans l'aide de chiffre? Et peut-on considérer ces observations comme des indicateurs de qualité des systèmes orientés objet?"*

Pour pouvoir parler d'indicateurs, il est nécessaire de donner à ces observations une méthode et un schéma strict à suivre. Nous parlerons dès lors de types de mauvaises utilisations de l'Orienté Objet observables. Dans les sections suivantes, nous développerons des méthodes pour observer les différentes caractéristiques orientées objet des systèmes. Nous montrerons par exemple comment observer le couplage général d'un programme. Ces différents *patterns visuels* observables formeront un "guide" de visualisation permettant de très vite pointer des particularités d'un système sans devoir se plonger dans des chiffres. Il est nécessaire de préciser que les types de mauvaises utilisations des principes orientés objet qui seront présentés plus tard, sont des propriétés de design illustrant une faiblesse au niveau couplage, cohésion ou héritage. Ce choix d'analyse résulte du fait qu'il est bien plus aisé de repérer des défauts plutôt que des qualités avec l'outil de visualisation.

La première observation nous a donc permis de dégager des types de défauts facilement observables. Nous nous sommes très vite rendu compte, après une première analyse de tout l'échantillon, que beaucoup de types de *patterns visuels* observables revenaient souvent au sein des architectures visualisées, par exemple, le *pattern visuel* représenté par un grand nombre de classes couplées dans un package isolé. C'est suite à cette constatation que l'idée de faire de toutes ces observations un guide de visualisation nous est venue. C'est grâce à ce guide que nous pourrons ensuite utiliser cette méthode de visualisation pour critiquer certaines qualités orientées objet d'un système. Nous allons donc définir des indicateurs visuels qui pointent chacun un type de défaut, un type de mauvaise utilisation de l'Orienté Objet que nous avons souvent observé.

3.2 Elaboration des indicateurs visuels

Dans cette section, nous verrons comme ont été élaborés les indicateurs visuels de mauvaise utilisation des principes orientés objet et de l'Orienté Objet en général.

3.2.1 Techniques d'observation utilisées

Dans cette section, nous expliquerons plus en détails de quelle façon nous observons les différents systèmes. Plusieurs observations de différents types sont pratiquées pour

pouvoir extraire un maximum de conclusions.

Nous parlerons souvent de couplage *général*, de cohésion *générale*, de complexité *générale* d'un système, ou encore d'utilisation *générale* du mécanisme d'héritage dans un système. Quand nous qualifions de *général* une propriété observable habituellement au niveau d'une classe, nous exprimons le fait que nous retirons de notre visualisation des idées et impressions qui nous permettent d'évaluer ces différentes caractéristiques au niveau du système entier. En effet, grâce aux techniques présentées ci-dessous, nous sommes capables de cerner à un plus haut niveau ces attributs qui sont habituellement des propriétés de classes. Donc, ces propriétés *générales* ne sont mesurées et obtenues par une quelconque formule entre les différentes valeurs de ces attributs pour chaque classe, mais simplement par estimation suite à l'observation visuelle et aux impressions ressenties.

Rappelons que les classes de types `Interface` ne sont pas représentées par le système car celles-ci pourraient influencer notre observation et introduire des informations inutiles qui fausseraient nos conclusions.

Nous utiliserons trois techniques principales de visualisation, représentées par la figure 3.1:

1. La première technique est utilisée pour observer de façon générale le système et sa structure, mais elle permet également, grâce à la navigation et toutes les autres possibilités offertes par le système, de pratiquer une observation plus détaillée. Cette première technique consiste à représenter toutes les classes et l'architecture en package sous la forme *Treemap*. Chaque boîte 3D est caractérisée par trois choses: sa taille (hauteur), sa couleur (variation de bleu vers rouge) et son orientation sur l'axe nord/sud (haut/bas). Pour la plupart des observations, cette technique sera utilisée en représentant la complexité cyclomatique d'une classe (WMCC) par la taille (non-observée pour le moment), le couplage par la couleur (la métrique CBO, plus la classe est rouge, plus le CBO est élevé), et la profondeur dans l'arbre d'héritage par l'orientation (la métrique DIT, plus la classe est horizontale, c'est-à-dire se rapproche de l'axe ouest/est, plus celle-ci se trouve profond dans un des arbres d'héritage). Nous associerons aussi souvent cette dernière caractéristique au manque de cohésion d'une classe (la métrique LCOM5, plus la classe est horizontale, c'est-à-dire se rapproche de l'axe ouest/est, moins celle-ci est cohésive).
2. La seconde technique est celle utilisée pour observer la structure des arbres d'héritage du système. La représentation est formée d'un disque divisé en portions. Chaque portion représente une profondeur dans l'arbre d'héritage et chaque portion contient les classes dont la valeur de DIT correspond. Nous pouvons de cette façon observer la répartition dans les arbres d'héritage même si cela ne nous permet pas de faire la différence entre les différents arbres. Dans cette visualisation, seule la couleur caractérise les classes, et plus elles sont rouges, plus elles se trouvent loin dans un arbre d'héritage (plus profond elles sont), ce qui nous permet de bien distinguer les différents niveaux des arbres d'héritage.

3. La troisième représentation est celle utilisée pour observer le nombre de classes peu ou pas cohésives dans le système tous packages confondus. Nous perdons de cette façon la représentation des packages mais nous avons une vue claire des proportions à observer. Nous pouvons appliquer un filtre sur cette représentation pour que les différentes valeurs de la métrique LCOM5 apparaissent encore mieux.

Grâce à ces trois techniques, nous pouvons observer les systèmes et repérer des types de mauvaises utilisations du code objet souvent rencontrés. Ces différentes techniques sont utilisées par l'observateur dans l'ordre qu'il souhaite et si il le souhaite. Il est évident que tous les filtres de visualisation que le programme met à notre disposition sont de temps à autre utilisés pour mettre certaines choses plus en évidence. De plus, le système d'observation offre encore beaucoup d'autres possibilités de mise en évidence de certaines valeurs ou proportions qui sont utilisées si nécessaire.

3.2.2 Types de mauvaises utilisations de l'Orienté Objet observables

Il est nécessaire de préciser ce que nous allons observer et surtout ce que nous serons capables de repérer grâce à ce système d'observation. Nous allons pouvoir, grâce aux exemples et indications qui suivent, facilement repérer ce que nous pouvons appeler des mauvaises utilisations de l'Orienté Objet et de ses principes et mécanismes. Il est évident que ce système permet de repérer beaucoup de choses, mais une liste précise de grands "défauts" observables sera développée par la suite pour guider l'observateur dans sa visualisation. Nous avons choisi de repérer des "défauts" car le système se veut très efficace pour accomplir cette tâche.

Les types de mauvaises utilisations des principes orientés objet et de l'Orienté Objet en général seront repérés grâce à l'observation de trois grandes caractéristiques mesurables de l'orienté objet: le couplage inter-classes, la cohésion des classes et la structure de l'arbre d'héritage. Dans les sections suivantes, nous développerons donc, pour chacune des ces trois caractéristiques, des types de mauvaises utilisations facilement repérables.

3.2.3 Observation du couplage

Il est nécessaire de préciser de quelle façon le couplage est observé, vu que cette métrique est en temps normal utilisée et observée au niveau d'une classe seule. Dans le cas présent, nous nous positionnons au niveau du système tout entier pour avoir une vue plus générale de la qualité du système dans son ensemble. Nous allons donc observer à ce niveau la concentration de classes fortement couplées dans une même partie du système, la mauvaise répartition du couplage dans les packages, le couplage exagéré de certaines classes isolées,... Ci-dessous sont rep*ri*ses les caractéristiques souvent observées durant l'analyse visuelle de l'échantillon. Ces caractéristiques constituent donc un partie de la panoplie des types de défauts que nous analyserons pour critiquer la qualité des systèmes

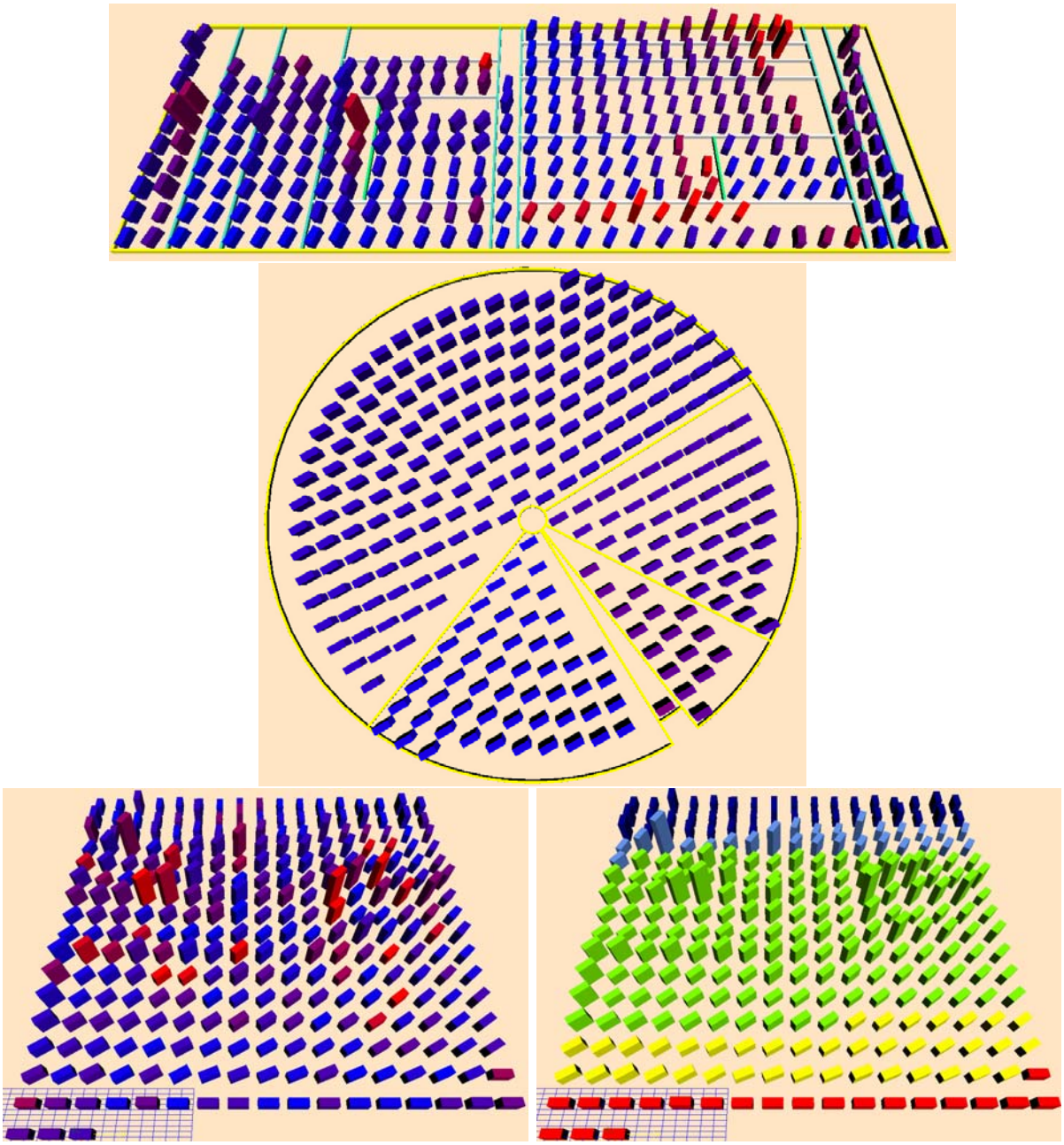

FIG. 3.1 – *Illustration (de droite à gauche et de haut en bas) des différentes techniques exposées dans la section 3.2.1, les différentes visualisations représentent le système "ProGuard". En premier, la représentation habituelle, ensuite la technique utilisée pour évaluer l'utilisation du mécanisme d'héritage, et enfin la technique utilisée pour observer la proportion de classes peu ou non cohésives dans le système. La dernière figure représente la même chose que la précédente mais avec un filtre de type "plot box" appliqué à la métrique LCOM5.*

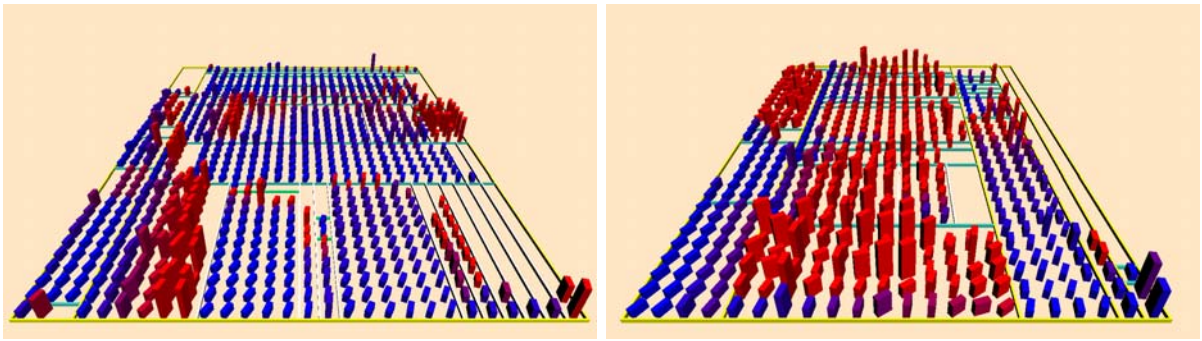

FIG. 3.2 – *Illustration de parties fortement couplées d'un système (à gauche le programme "PCGen", et à droite "Art Of Illusion"). Les parties visées sont reconnaissables par la profusion de classes représentées dans un rouge très vif.*

objet.

Rappelons que, dans l'outil de visualisation, nous représentons le couplage par la variation de la couleur bleue vers la couleur rouge. Plus une classe est rouge, plus le couplage de celle-ci est élevée (métrique CBO).

Chaque section suivante détaille une caractéristique observable. Ces représentations détaillées en font des types de mauvaises utilisations de l'Orienté Objet, qui pénalisent entre autres, la maintenabilité d'un logiciel.

Des parties du système fortement couplées

Dans beaucoup de systèmes observés, nous pouvons remarquer que certaines parties (package, plusieurs packages, un groupe de classes,...) d'un système apparaissent comme formées de classes fortement couplées. Dans une majeure partie des cas, le couplage est concentré dans ces parties, alors que le reste du système présente une assez bonne répartition du couplage. Cette observation nous permet donc de penser que la répartition des tâches dans les classes a peut-être été mal pensée vu la forte interdépendance entre certaines classes. Nous qualifions cette caractéristique de mauvaise habitude orientée objet, parce que, dans le contexte d'une maintenance, ce fort couplage concentré peut amener de très nombreuses et fastidieuses modifications alors que la correction de départ est minime. Nous pouvons voir deux exemples de ce type de mauvaise utilisation de l'Orienté Objet à la figure 3.2. Les deux systèmes représentés possèdent certains packages caractérisés par un très fort couplage général.

Une mauvaise répartition du couplage

Cette caractéristique se représente la plupart du temps, au sein d'un package, par une classe fortement couplée à d'autres, alors que les autres classes du package présentent un couplage faible. Visuellement, cela est représenté par de rares classes très rouges dans

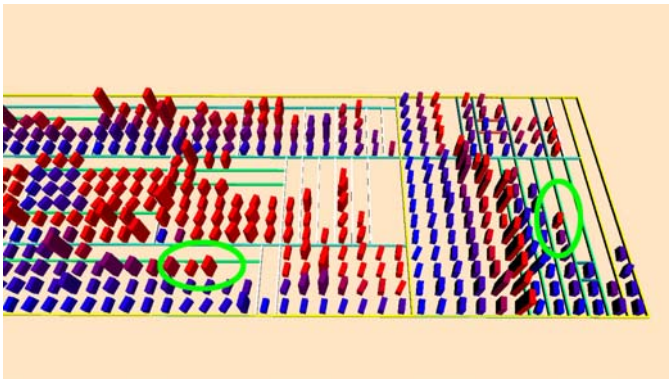

FIG. 3.3 – *Illustration d'une mauvaise répartition du couplage dans un système (le programme représenté est "Merchant Of Venice"). Les zones visées sont mises en évidence et entourées par une marque verte.*

un package alors que le reste des classes de ce package tend plutôt vers le bleu. Nous pouvons en déduire que le couplage est mal réparti, et donc que la répartition des tâches au sein des classes est peut-être à revoir. Nous considérerons donc ce phénomène comme un défaut dans un code orienté objet. Nous pouvons estimer que cette classe rassemble beaucoup d'applications et beaucoup de méthodes qui n'ont pas vraiment leur place au sein de celle-ci. Cela reflète donc une mauvaise qualité orientée objet car la compréhension du système n'est pas aisée quand celui-ci est composé de classes de ce type. De plus, la maintenabilité du système est de piètre qualité dans ce cas, car il est difficile de pratiquer une modification d'une classe qui en influence beaucoup d'autres, au risque de devoir faire des modifications en chaîne. Un exemple de ce type de mauvaise utilisation de l'Orienté Objet est illustré par la figure 3.3. Nous remarquons en effet des classes très couplées par rapport au reste des classes du même package.

Des classes isolées et extrêmement couplées

Comme dit précédemment, on remarque de temps en temps des classes au couplage très élevé. Parfois ces classes sont isolées, seules dans un package. Cette structure trahit une mauvaise habitude de codage, qui consiste à créer une classe désordonnée contenant les méthodes et variables que le programmeur n'a pas réussi à intégrer au sein de son architecture. Nous remarquons dans ce cas qu'une sous-structure de packages a été mise en place uniquement pour cette classe, ce qui trahit soit un oubli suite à une restructuration, soit une structure mal pensée au départ et donc la création de cette classe sans essence propre. Un système présentant ce défaut est visible à la figure 3.4. Ici, nous pouvons observer une classe complètement isolée présentant un couplage extrême.

3.2.4 Observation du mécanisme d'héritage

Pour observer l'héritage avec l'outil de visualisation, nous répartissons les classes dans des zones en fonction de leur profondeur dans l'arbre d'héritage (DIT). La variation de

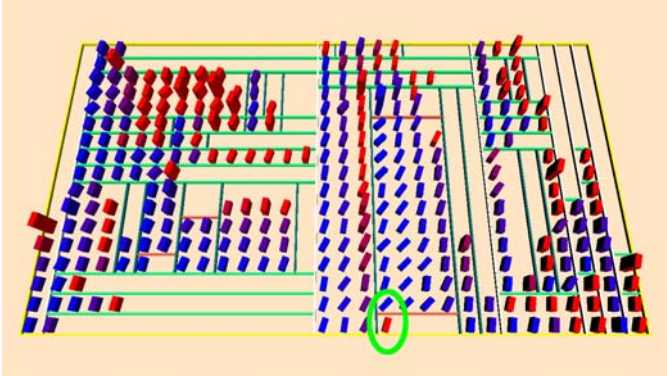

FIG. 3.4 – *Illustration de classes isolées dans le système et qui sont fortement couplées (le programme représenté est "Mantaray". La zone visée est mise en évidence et entourée par une marque verte.*

DIT est représentée ici par la variation de bleu vers rouge. Sur les images, la portion au bas de l'image représente le niveau le plus haut (le plus proche de la racine) d'un arbre d'héritage. Ensuite, il suffit d'observer le diagramme dans le sens des aiguilles d'une montre pour descendre dans les différents niveaux des structures d'héritage. Il est évident que toutes ces classes ne sont pas dans le même arbre, et que cette visualisation ne fait pas la différence entre les différents arbres. Mais avec un tel outil, nous pouvons repérer la répartition de l'héritage en général dans tout le système.

Nous pouvons aussi observer les structures d'héritage utilisées de façon plus classique. En donnant par exemple à une des propriétés des boîtes trois dimensions l'ordre de représenter la valeur de DIT. Ce qui nous permet de voir dans quels packages l'héritage est fortement utilisé ou le contraire. Dans notre cas, nous choisirons l'orientation de la boîte 3D sur l'axe haut-bas pour représenter la profondeur dans l'arbre d'héritage.

Mauvais équilibrage de l'arbre d'héritage

Il est aisé de remarquer une concentration d'un grand nombre de classes dans un seul niveau de l'arbre d'héritage. Il est également facile de repérer si les arbres d'héritage sont plutôt profonds ou plutôt plats. Nous critiquerons ici uniquement l'allure générale de l'utilisation du mécanisme d'héritage. Nous critiquerons surtout un système avec de l'héritage mal utilisé plutôt qu'un système presque sans mécanisme d'héritage, car cela ne signifie par nécessairement qu'il existe un problème au niveau de la maintenabilité du système. A la figure 3.5, nous pouvons observer deux systèmes composés d'arbres d'héritage mal équilibrés.

Des classes isolées très basses dans l'arbre d'héritage

Un propriété souvent observée est le fait qu'une classe se retrouve seule dans une feuille, et que toute une architecture d'héritage ait été mise en place pour cette classe

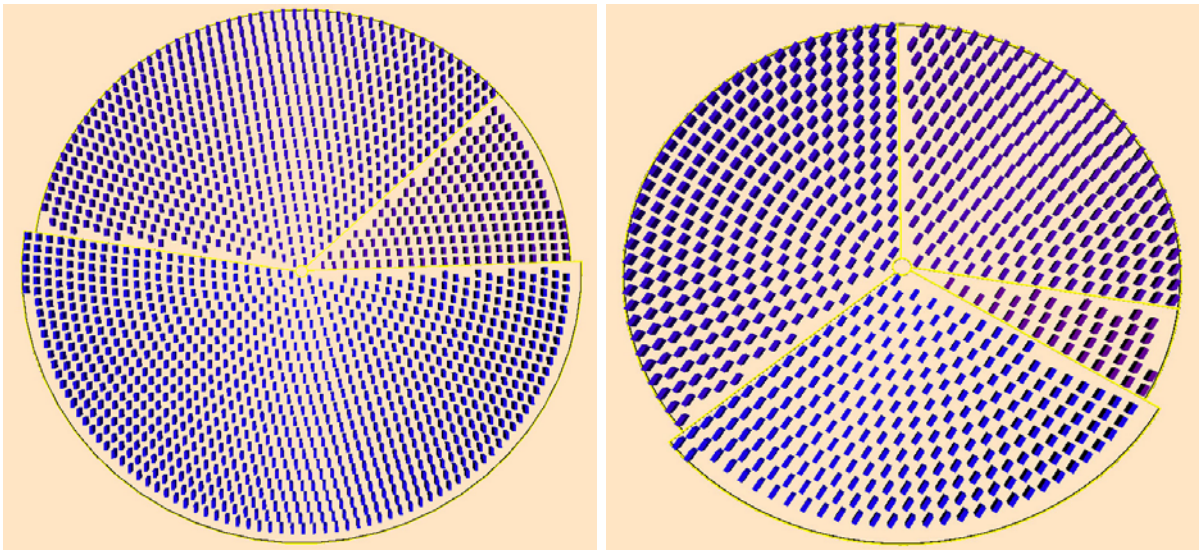

FIG. 3.5 – *Illustration de deux systèmes dont les arbres d'héritage paraissent très plats et peu équilibrés (les programmes représentés sont: à gauche "Liferay Portal" et à droite "PCGen".*

unique, alors que l'héritage est très peu utilisé dans le reste du système. Cela trahit, soit un oubli lors d'une refonte de l'architecture, soit une architecture peu homogène mise en place à la création du système. Cette observation sera aussi considérée comme un problème car une telle architecture imposera au programmeur, lors de la maintenance, de devoir se plonger plus profondément dans la compréhension du système à cause de tels arbres d'héritage la plupart du temps inutiles. Nous pouvons observer sur la figure 3.6 deux représentations possibles nous permettant de repérer ce type de mauvaise utilisation des mécanismes de l'Orienté Objet, ici celui d'héritage. La seconde représentation illustre la technique habituelle de visualisation couplée à l'utilisation d'un filtre de type *Plot Box* sur les valeurs de DIT.

3.2.5 Observation de la cohésion générale du système

La cohésion, autre métrique orientée objet mesurée au niveau d'une classe, sera ici observée pour le système en général. Nous observerons des ensembles de classes pour pouvoir décider de la cohésion générale du système. Une domination de classes peu, voire non cohésives au sein d'un même package, sera, par exemple, signe d'une très mauvaise cohésion générale d'une partie du système observé.

Rappelons que dans le cas de la visualisation de la cohésion, la métrique LCOM5 de chaque classe est représentée par son orientation par rapport à l'axe nord/sud de la structure représentée. Plus l'orientation de la classe se rapproche de l'axe ouest/est, plus la valeur de la métrique LCOM5 se rapproche de 2, c'est-à-dire, moins cette classe est cohésive.

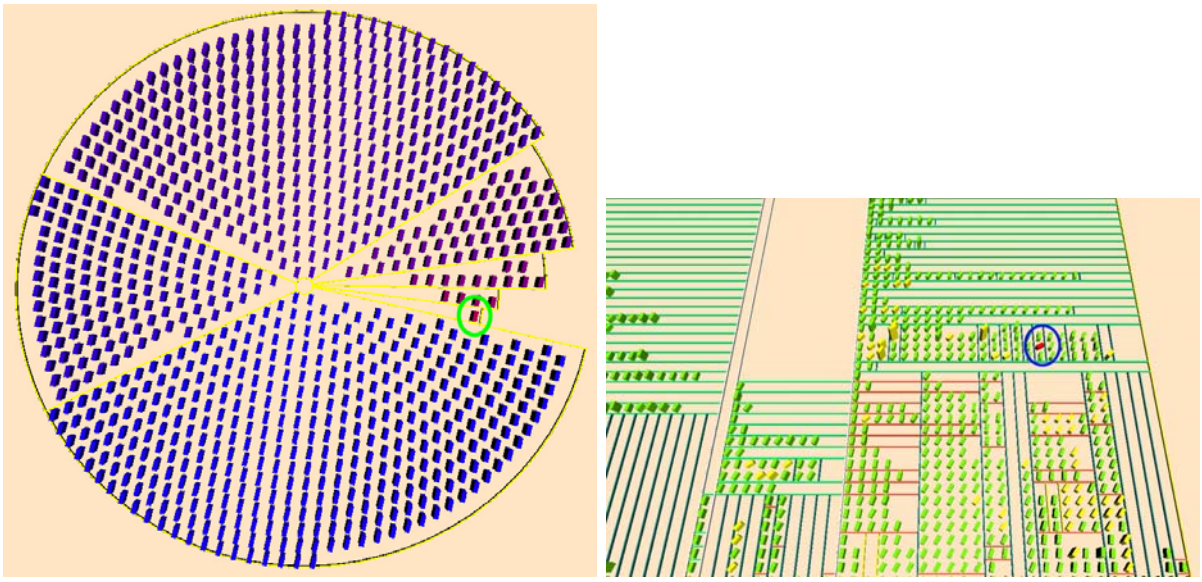

FIG. 3.6 – *Illustrations du système "Columba" dans lequel nous retrouvons une classe isolée très profonde dans l'arbre d'héritage. La classe visée est entourée de vert sur l'image de gauche et de bleu sur l'autre.*

Très mauvaise cohésion générale de certains packages

Comme cité ci-dessus, une mauvaise cohésion de certaines parties du système due à une très mauvaise cohésion de la majorité des classes de ces parties, sera considérée comme un signe d'une mauvaise qualité. Cela se justifie tout simplement parce qu'une mauvaise cohésion entraîne évidemment une difficulté de maintenance du système. Le manque de cohésion est un principe contraire à l'Orienté Objet qui tend à rassembler dans une classe tout attribut ou méthode qui a directement un rapport avec l'objet représenté par la classe. Nous pouvons observer ce type de mauvaise utilisation à la figure 3.7. En effet, cette figure représente un programme contenant un package dont les classes sont très peu cohésives. Celles qui sont le moins cohésives apparaissent en jaune ou en rouge. En analysant les valeurs de plus prêt, il est possible d'affirmer cette intuition.

Des classes isolées pas du tout cohésives

Le cas d'une classe isolée dans le système se montrant très peu cohésive ou pas cohésive du tout est similaire au cas d'une classe isolée présentant un couplage fort. Cela trahit une architecture mal pensée et indique que cette classe est certainement une classe créée par besoin, qui contient des méthodes et attributs sans rapport entre-eux. C'est de là que vient souvent le manque certain de cohésion d'une classe isolée. Celle-ci est souvent accompagnée d'un fort couplage. Nous voyons sur la figure 3.8 une classe complètement isolée dans un package caractérisée par une cohésion nulle. Nous pouvons sans doute conclure que cette classe est une classe sans cohérence interne qui a été créée pour pallier à d'autres problèmes. Cette solution est une solution de facilité mais très rarement une

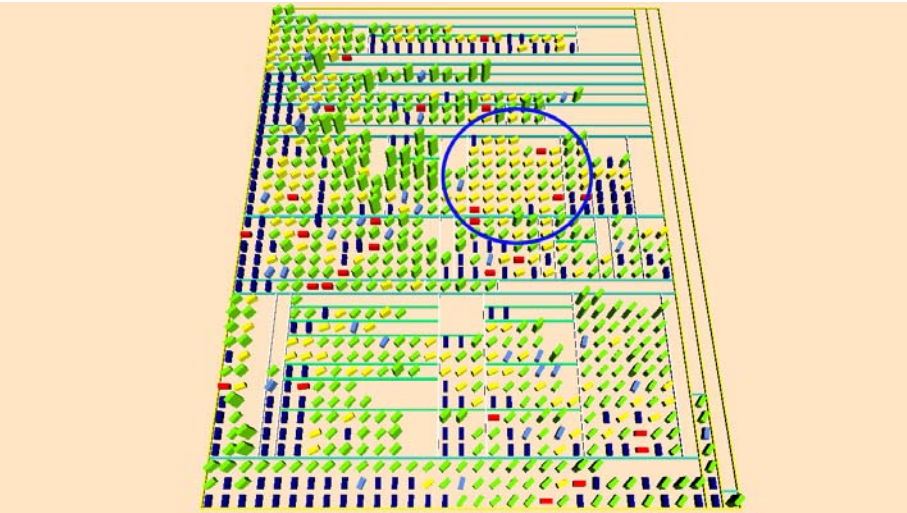

FIG. 3.7 – *Illustration du système "Freenet" (sous un filtre mettant en évidence les valeurs de LCOM5) dans lequel nous retrouvons un package dont la cohésion générale est très mauvaise. Celui-ci est mis en évidence par une marque bleue.*

bonne solution car le fait de pratiquer cette méthode complique fortement la compréhension du système (vu que certaines classes n'ont pas de "rôle" particulier et bien défini) et donc sa modification future.

Majorité de classes peu ou pas cohésives dans le système entier

La dernière mauvaise habitude orientée objet que nous pouvons remarquer concerne tous les systèmes présentant une très mauvaise cohésion générale. On remarque donc qu'aucun effort de réflexion approfondie n'a été pratiqué lors de la répartition en différents objets. Pour pouvoir observer ce style de problème, nous pouvons pratiquer un autre type d'observation, en classant les objets 3D par leur cohésion, comme expliqué dans la section décrivant les techniques d'observations. Pour ce type de mauvaise utilisation, nous reprendrons l'exemple du système "Freenet" que nous retrouvons à la figure 3.9. Ici, le programme est représenté sans sa structure de packages. Les classes sont triées selon leur valeur de LCOM5 et l'orientation nord/sud des classes représente leur valeur de LCOM5. Nous trouvons au bas de l'image et en jaune ou rouge, les classes peu ou non cohésives. Cette visualisation nous permet donc de voir la proportion des classes présentant une cohésion acceptable ou bonne. Après une analyse plus détaillée de cette représentation, nous pouvons dire qu'environ la moitié des classes du système sont peu ou pas cohésives.

3.3 Méthode d'interprétation des indicateurs

Maintenant que nous avons développé différents types de mauvaises utilisations de l'Orienté Objet facilement observables avec notre système, il serait intéressant de voir en

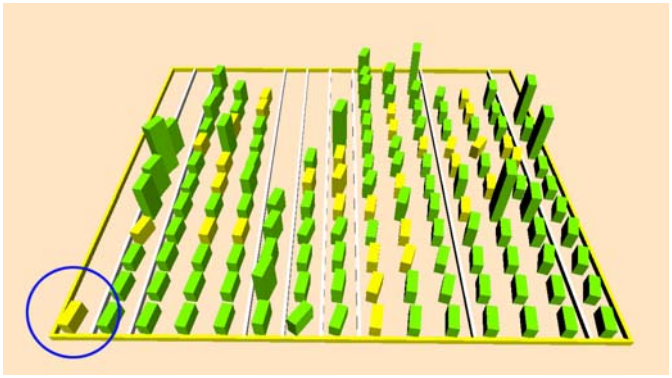

FIG. 3.8 – *Illustration du système "Bots'n'Scouts" (sous un filtre mettant en évidence les valeurs de LCOM5) dans lequel nous retrouvons une classe isolée avec une valeur de LCOM5 égale à 2, ce qui signifie que la cohésion de cette classe est nulle. Cette classe est mise en évidence par une marque bleue.*

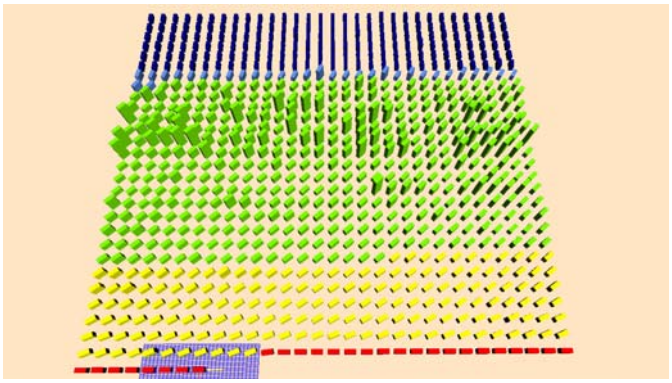

FIG. 3.9 – *Illustration du système "Freenet", selon la troisième technique de visualisation présentée dans la section 3.2.1 (page 58), dans lequel nous retrouvons une forte proportion de classes très peu cohésives.*

quoi ces types de faiblesses peuvent être des indicateurs et surtout comment nous devons les interpréter.

Nous pouvons dire intuitivement que ces différents points observables sont des indicateurs. Ils indiquent chacun une certaine faiblesse à un certain niveau du système. Cette faiblesse se fera très souvent ressentir lors de la maintenance du système. Nous pouvons donc presque dire que ces indicateurs sont des indicateurs de bonne ou mauvaise maintenabilité des systèmes orientés objet. Toutefois, comme nous l'avons brièvement dit, même si nous pouvons repérer ces "défauts", ces faiblesses intuitives, il ne s'agit pas automatiquement d'un mal. Il est évident que certaines techniques de programmation et une documentation sont cachées derrière ces observations et que le fait d'observer une répartition dans les arbres d'héritage peu homogène ne veut pas automatiquement dire que le système fait preuve d'une mauvaise maintenabilité. Il est donc nécessaire d'interpréter ces indicateurs ensemble et non de façon isolée.

Pour pouvoir tirer des conclusions grâce à cette technique d'observation, il faut donc analyser le système selon les trois grandes caractéristiques des systèmes orientés objet présentées, à savoir le couplage, la cohésion et l'utilisation du mécanisme d'héritage. Une fois une première idée du système acquise, il faut ensuite observer chacune de ses caractéristiques plus en détails. C'est-à-dire qu'à ce moment, nous devons utiliser les types de faiblesses observables que nous venons d'exposer. Ces différents repérages de types de mauvaises utilisations de l'Orienté Objet nous permettent d'évaluer le couplage général du système, la cohésion générale du système ainsi que l'utilisation forte ou faible, cohérente ou non, homogène ou non du mécanisme d'héritage. L'évaluation de chacun des trois points se fera donc suite au repérage ou non de certaines faiblesses décrites plus haut. Nous évaluerons aussi la gravité de ces mauvaises utilisations, car il est évident que, par exemple, un système avec beaucoup de classes couplées, ne doit pas nécessairement être jugé de la même façon qu'un autre système présentant la même faiblesse dans une moindre mesure. Une fois toutes ces observations pratiquées et toutes ces informations extraites, nous pouvons donner un jugement au système concernant sa qualité orientée objet. Encore un fois, précisons que nous jugeons surtout un attribut de la qualité logicielle, ici la maintenabilité. En croisant toutes les informations récupérées grâce aux observations, nous pouvons évaluer cet attribut de qualité et donc utiliser cette méthode comme un indicateur général de la qualité orientée objet du système, en tous cas pour ce qui concerne la maintenabilité du système.

Nous pouvons conclure cette section en donnant un exemple. Nous ferons ici une brève critique du système "Azureus" (logiciel de partage de fichiers "peer to peer") selon la méthode que nous venons d'exposer. Les figures 3.10 et 3.11 nous illustrent le système à analyser selon les différentes techniques de visualisation énoncées dans la section 3.2.1 (page 58). La figure 3.10 illustre la première technique de visualisation de deux façons différentes (représentation en *Treemap*). En premier, la valeur de la métrique DIT est représentée par l'orientation des classes en trois dimensions, et, en second, la valeur de

LCOM5 est représentée par cette caractéristique. La seconde visualisation est accompagnée d'un filtre pour mettre en évidence les différentes classes de valeurs de LCOM5. La seconde figure (3.11) représente les deux autres techniques, à savoir celle permettant d'observer la répartition dans les arbres d'héritage et celle permettant d'analyser la proportion de classes peu ou pas cohésives dans un système. Voici la critique que nous pouvons faire de ce système grâce à l'outil de visualisation et au "guide" constitué des types de mauvaises utilisations observables:

- Ce système semble à première vue être un bon système orienté objet. Tout d'abord, on remarque une bonne utilisation du mécanisme d'héritage. On déduit des valeurs de DIT observées une bonne répartition des classes dans l'arbre d'héritage, sans trouver de valeur trop extrême. L'héritage semble utilisé de façon assez homogène et pas exceptionnellement.
- En ce qui concerne le couplage: au premier regard, on remarque un couplage faible en observant que la plupart des classes sont colorées en bleu (la valeur de CBO est représentée par une variation de couleur du bleu vers le rouge). Quelques classes affichent un taux de CBO extrême mais elles sont très rares. On peut donc dire que le couplage général du système est très faible.
- La cohésion est sans doute le point un peu plus faible de ce système. La valeur de cette métrique n'est pas très constante. Cependant, il faut souligner que deux tiers du programme affichent une valeur de LCOM5 inférieure à 1. Les classes vraiment peu cohésives sont rares.

Suite à cette brève analyse, nous avons conclu que "Azureus" était un bon système orienté objet car nous n'avons pas vraiment pu repérer de type de mauvaise utilisation de l'Orienté Objet. Il y a évidemment quelques imperfections mais il est certain que nous ne pouvons pas prendre ce système comme exemple de ce qu'il ne faut pas faire en Orienté Objet.

3.4 Validation empirique des indicateurs

Comme nous l'avons exprimé dans le chapitre présentant la méthodologie que nous désirions suivre tout au long de cette recherche, notre souhait est évidemment que ces indicateurs et cette méthode soit reconnus comme une technique permettant de juger avec une certaine fiabilité (selon l'échantillon et l'observateur) certaines qualités orientées objet d'un système.

Pour le moment, plusieurs observateurs experts en génie logiciel se sont penchés sur la méthode. Ces personnes encadraient la recherche effectuée, ou même travaillaient dans le même domaine en exploitant les mêmes outils. Les résultats prometteurs que nous avons obtenus les ont intrigués et ils ont donc voulu participer à la validation de cette technique d'observation. Leurs avis concordaient avec les nôtres et la technique décrite ici s'avérait même être une synthèse assez fournie de ce que d'autres utilisateurs de ce système de

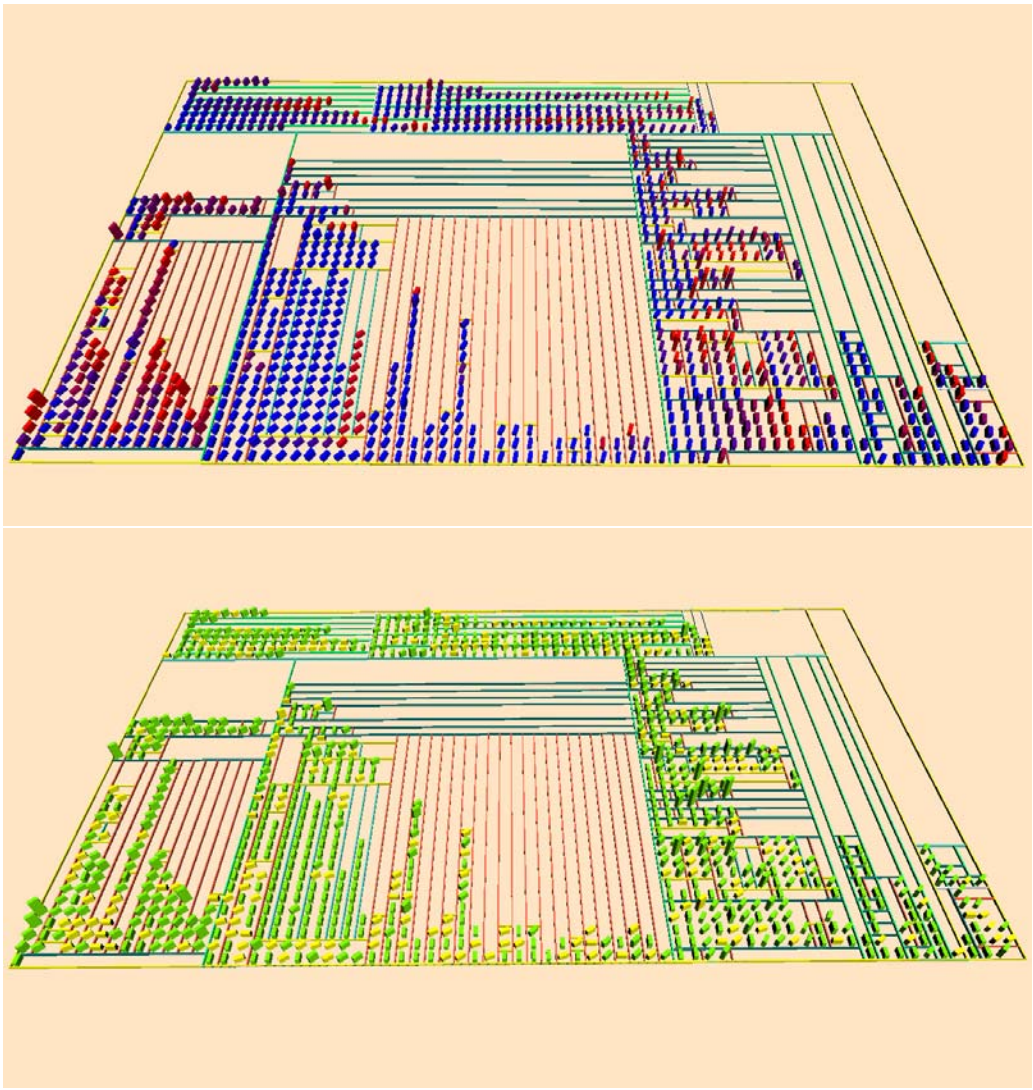

FIG. 3.10 – *Illustration du système "Azureus", selon la première technique de visualisation présentée dans la section 3.2.1 (page 58).*

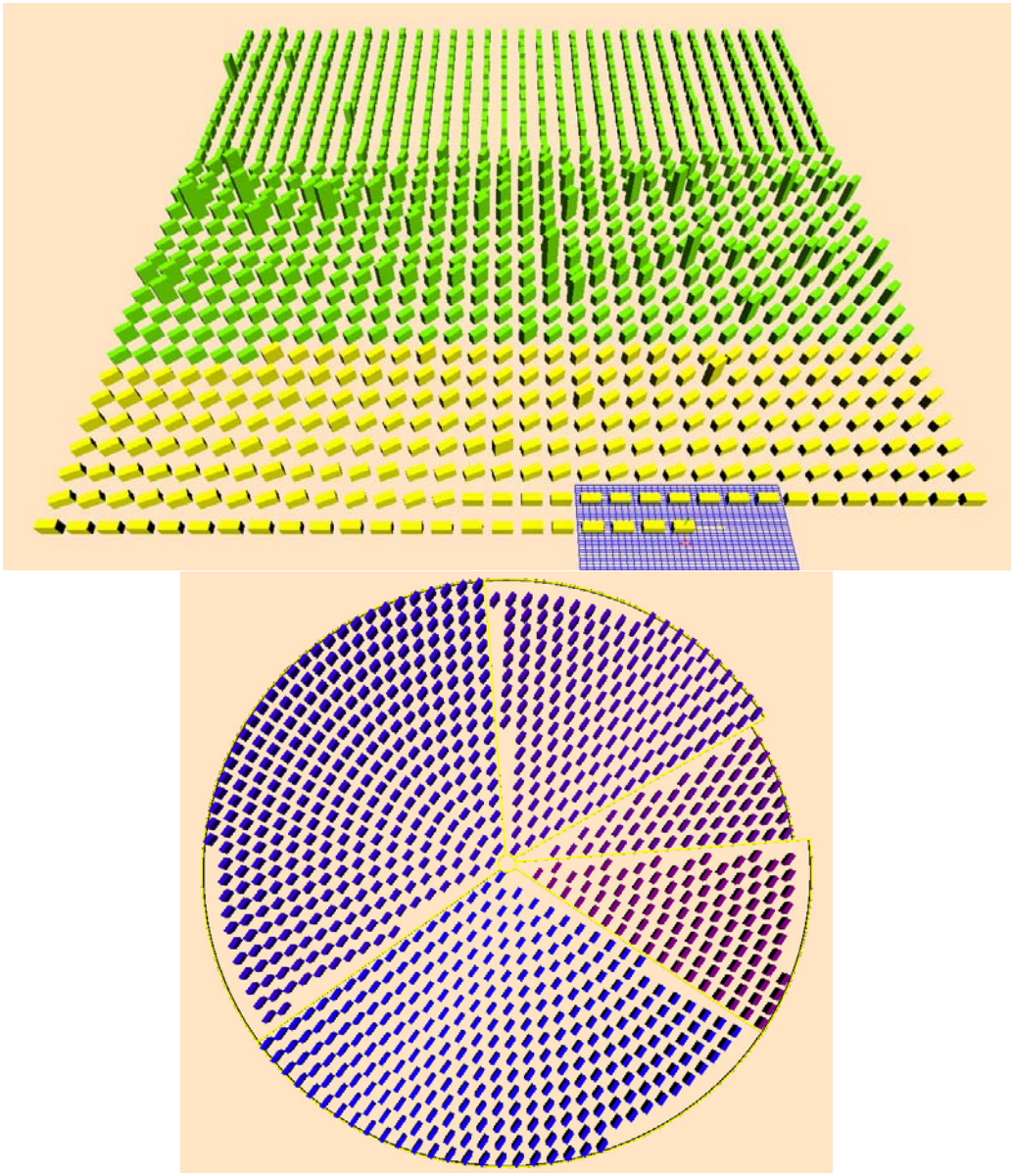

FIG. 3.11 – *Illustration du système "Azureus", selon la seconde et la troisième technique de visualisation présentées dans la section 3.2.1 (page 58).*

visualisation utilisaient comme indicateurs sans les avoir définis comme tels.

Cette constatation nous permet donc de dire que ce que nous sommes capables de mettre en évidence avec cette technique d'observation et d'évaluation peut constituer une panoplie d'indicateurs pour évaluer certaines qualités des systèmes orientés objet comme la réutilisation et la maintenabilité. Il est évident qu'il serait valorisant et intéressant de pouvoir présenter ces résultats à une plus grande communauté pour que la validation de ces indicateurs acquiert une plus grande crédibilité. Cette étape n'a pas pu encore être réalisée mais nous présenterons dans la section "Travaux futurs" la méthode qui pourra être suivie pour lancer le processus de validation à plus grande échelle.

3.5 Utilisation des indicateurs pour tester l'utilité de la complexité cyclomatique

Dans cette section, nous allons montrer comment nous sommes arrivés à utiliser ces indicateurs et ce système de visualisation pour pouvoir évaluer la complexité générale du système. De plus, nous montrerons comment nous avons pu obtenir des résultats pour venir affirmer ou infirmer notre deuxième hypothèse qui a été formulée de cette façon: *"La complexité cyclomatique, analysée au niveau du système entier, peut, à elle seule, constituer une indicateur suffisamment fiable de bonne utilisation des principes orientés objet"*.

3.5.1 Observations multiples et tests

Comme nous l'avons déjà expliqué et montré sur certaines illustrations, la complexité cyclomatique d'une classe (mesure WMCC) peut être représentée par la taille en hauteur des objets trois dimensions représentant les classes. Tout au long du processus d'observation suivi pour pouvoir réaliser un guide de visualisation nous permettant de pouvoir évaluer certaines qualités orientées objet d'un système, nous avons pu remarquer certaines choses. Nous avons très souvent constaté, sans approfondir la recherche, que quand nous observions un système dont nous pouvions dire, grâce à notre méthode, que certaines qualités orientées objet de celui-ci laissaient à désirer, nous pouvions observer différents phénomènes, différents motifs visuels (*patterns visuels*) liés à la complexité cyclomatique. En général, plus un système est "mauvais", c'est-à-dire, affiche une cohésion générale mauvaise, en plus d'un couplage général élevé et d'une incohérence du mécanisme d'héritage, plus celui-ci affiche une complexité élevée. Il est très rare d'observer le contraire. Nous avons donc essayer de valider cette hypothèse de plusieurs façons.

Suite à ces observations multiples, nous avons essayé de voir si le couplage, la cohésion ou le mécanisme d'héritage était, seul à seul, directement lié à la complexité cyclomatique d'un classe seule, et puis à la complexité cylomatique générale du système. Rappelons encore une fois que quand nous parlons de complexité cyclomatique générale ou de cou-

plage général, il ne s'agit nullement d'une somme ou de quelconques opérations entre les différentes valeurs, mais simplement d'une observation visuelle générale de cette caractéristique au niveau du système entier.

Corrélation 2 à 2 au niveau de la classe

Pour atteindre le premier objectif, c'est-à-dire chercher un lien direct entre différentes caractéristiques d'une classe, nous avons simplement recherché des corrélations directes entre les valeurs caractérisant une classe (valeur de DIT, de CBO, de LCOM5 et de WMCC), ces valeurs sont obtenues grâce à l'outil POM. Cette recherche de corrélation a été faite de façon numérique grâce à des outils de statistiques mais aussi grâce à l'analyse visuelle de classes individuelles. Après plusieurs essais, nous n'avons pas pu trouver de corrélations directes entre le couplage, la cohésion, l'héritage et la complexité pris deux à deux au niveau d'une classe seule. Il est d'ailleurs aisé de trouver un grand nombre de contre-exemples.

Corrélation 2 à 2 au niveau du système entier

Le second objectif était de voir si, en observant ces caractéristiques pour le système entier (plus de valeurs, mais juste l'idée générale que nous fournit la visualisation), nous pouvions trouver une corrélation entre le couplage général et la complexité générale ou entre l'utilisation du mécanisme d'héritage et la complexité générale ou entre la cohésion générale et la complexité générale. Nous avons donc repris nos observations mais nous sommes tombés pour chacune des possibilités sur des cas contradictoires. Nous concluons donc qu'aucune des caractéristiques générales du système orienté objet prise seule ne varie de façon liée à la complexité cyclomatique générale du système.

Cependant, nous restions convaincus que cette nouvelle façon d'observer un système orienté objet nous ouvrait une nouvelle porte concernant la complexité cyclomatique. De plus, certains écrits de la littérature montrent que d'autres recherches sont menées à propos du fait que la complexité cyclomatique pourrait être un indicateur de qualité orientée objet. Nous avons donc pensé que la complexité cyclomatique du système dépendait de plusieurs des caractéristiques à la fois.

Plusieurs solutions s'offraient alors à nous. Soit nous procédions, avec des outils de statistiques, à une recherche de corrélation entre des valeurs, soit nous continuions à essayer de trouver des résultats grâce à l'outil de visualisation. La première solution ne nous a mené à rien, tout d'abord parce qu'il était impossible de savoir comment présenter les données pour trouver une corrélation intéressante. Devions-nous calculer des régressions linéaires et autres calculs sur toutes les données de tous les systèmes à la fois ou sur un seul système à la fois? Devions-nous discrétiser les variables, les présenter d'une autre façon? Ces questions et bien d'autres ont bloqué notre recherche dans ce sens et comme nous avions l'outil de visualisation sous la main et une certaine expertise acquise, nous avons penché pour la seconde solution. De plus, c'est grâce à cet outil que nous avons eu

cette intuition et il était donc plus intéressant et plus facile d'arriver à la démontrer par les mêmes moyens dans un premier temps.

Recherche par analyse visuelle

Le problème que pose la visualisation est le fait que le jugement donné reste, d'une certaine manière, subjectif. Pour pallier à ce problème, nous exigerons que l'évaluation de chaque caractéristique orientée objet observée soit donnée uniquement sur base d'observation de types de mauvaises utilisations de l'Orienté Objet et de ses mécanismes décrits précédemment. Cette exigence est nécessaire pour essayer d'arriver à une évaluation la plus objective possible et plus ou moins commune à tous les observateurs.

Un autre problème qui se pose à ce niveau est qu'il est nécessaire de chiffrer ou de classifier nos analyses visuelles pour, plus tard, essayer une ou plusieurs corrélations entre nos évaluations du couplage général, de la cohésion générale, de l'utilisation de l'héritage dans le système et de la complexité générale du système. Nous allons donc, avant de confronter nos données aux recherches statistiques de corrélation, exposer une proposition de méthode permettant de passer d'une évaluation qualitative d'un attribut du système à une valeur chiffrée de cette évaluation.

A l'instar d'un juge observant la prestation d'un patineur lors d'une compétition de patinage artistique, nous allons attribuer plusieurs scores à un système suite à son analyse visuelle (un score par attribut: couplage général, cohésion générale, utilisation de l'héritage et enfin complexité générale). Comme le juge de patinage artistique, notre observation sera guidée afin de pouvoir, finalement, être résumée en scores ou cotations. Nous allons donc focaliser notre analyse sur plusieurs points critiques. Ce canevas d'observation permet de rester les plus objectif possible, il est constitué du "guide" de visualisation que nous avons élaboré préalablement. Nous allons utiliser toutes les techniques de visualisation présentées et les différents défauts d'un logiciel observables pour juger, cette fois quantitativement, un logiciel.

Il est nécessaire maintenant de décrire la méthode que nous allons concrètement utiliser sur notre échantillon. Cette méthode se basera sur un système de cotation. Chaque caractéristique orientée objet sera jugée grâce à la méthode que nous avons élaborée et un score lui sera attribué, sur une échelle de 0 à 10 dans ce cas-ci. Les scores seront aussi donnés en comparaison à des systèmes que nous considérons comme bien programmés, tels le système "Azureus" présenté précédemment. Les scores attribués ne dépendent donc que de ce que nous pouvons visuellement et qualitativement évaluer et non de formules entre différentes valeurs.

Nous procéderons en trois étapes:

1. La première étape consiste en l'observation du **couplage** général du système, de la **cohésion** générale du système et de l'utilisation du mécanisme d'**héritage** dans le

système en entier. L'observation se fait sur base des types de faiblesses observables et ensuite, un score est donné à chacune de ces caractéristiques.

2. La seconde étape consiste à pratiquer une évaluation similaire mais pour la **complexité cyclomatique** générale du système. Pour cette caractéristique, nous n'avons pas élaboré de guide, mais il serait très semblable à celui donné pour l'évaluation de la cohésion générale du système (classe isolée très complexe, complexité générale élevée, complexité d'un package extrême). Il suffit ensuite de donner un score à la complexité. Il est évident que la première et la seconde étape doivent se faire indépendamment l'une de l'autre pour éviter toute influence. Il est d'ailleurs possible, avec le système de visualisation, de ne représenter que la complexité cyclomatique pour ne pas être influencé dans l'évaluation.
3. La dernière étape consiste, une fois les 4 scores donnés pour chaque système, à trouver une corrélation entre toutes ces valeurs. Dans notre cas, nous avons pensé, selon notre intuition de départ, qu'une combinaison linéaire des trois premiers scores (concernant le couplage, la cohésion et l'héritage), pouvaient influencer, ensemble, le quatrième.

Ce canevas a été suivi pour l'analyse de notre échantillon. Nous avons procédé à la recherche de corrélation une fois tous les scores attribués aux systèmes de notre échantillon et nous avons obtenu quelques premiers résultats intéressants et prometteurs (la fiabilité étant celle de notre système de visualisation et de notre échantillon). Ces premiers résultats, que nous présenterons en conclusion de ce travail, nous poussent à penser que notre seconde hypothèse est correcte. Nous avons en effet pu montrer, avec notre échantillon, qu'un certain lien existait entre la complexité cyclomatique observée de façon générale et nos trois caractéristiques orientées objet également observées d'un plus haut niveau que celui de la classe seule. Nous avons donc tendance à dire que: moins certaines qualités orientées objet sont bonnes (moins certains attributs de la qualité logicielle sont bons), plus la complexité générale du système sera grande. La complexité cyclomatique observée pour le système en général pourrait donc être un indicateur à elle seule d'un "bon" ou "mauvais" système orienté objet, c'est-à-dire d'un système qui utilise bien ou non les mécanismes orientés objet.

3.5.2 Validation empirique de la complexité cyclomatique comme indicateur

Nous pouvons, maintenant que nous avons mis une technique au point, essayer de faire valider cette technique et surtout les résultats correspondants. Plusieurs utilisateurs de notre système de visualisation se sont penchés sur les résultats obtenus qui tendent à montrer que seule la complexité cyclomatique serait suffisante pour indiquer si un système est bien programmé en Orienté Objet ou non. Ces résultats que nous avons obtenus constituent une première validation de type mathématique. En effet, cette validation a été faite sur base de recherche de corrélations entre des données chiffrées obtenues par évaluation visuelle. Nous verrons plus tard les données obtenues grâce à notre méthode de

scores permettant d'effectuer le passage entre l'idée visuelle obtenue par observation à des données chiffrées nous permettant de faire des recherches de corrélation précises et rapides.

La recherche effectuée se limite bien entendu au nombre de personnes qui ont participé, mais aussi à notre échantillon de systèmes orientés objet. L'étape suivante serait donc, dans un premier temps, de présenter notre méthode de visualisation et d'évaluation à des experts tiers, et de leur demander de pratiquer une évaluation de notre échantillon selon notre méthode. Un fois ces scores obtenus, nous pourrions nous mettre à la recherche de corrélation pour chacune des évaluations différentes d'experts différents, ou encore, nous pourrions calculer une moyenne des scores donnés pour dresser un tableau de scores général et rechercher des corrélations sur base de ces données. Avec ces nouveaux scores, il suffirait de voir si les résultats de corrélation sont aussi intéressants ou s'ils viennent contredire ce que nous pensons après notre analyse de notre échantillon.

La seconde étape sera de faire les mêmes recherches mais sur un autre échantillon ou simplement sur le même mais enrichi de nouveaux systèmes pour s'assurer que nos résultats ne proviennent pas d'une exception due à un choix trop restrictif ou subjectif des programmes analysés.

Ces deux étapes n'ont pas pu être réalisées dans le cadre de ce travail mais pourraient évidemment apporter une valeur ajoutée importante à nos recherches, que les résultats soient positifs ou non.

Chapitre 4

Conclusion

4.1 Résultats obtenus

Comme nous l'avons montré dans le chapitre précédent, notre recherche nous a conduit à certains résultats intéressants.

Premièrement, pour ce qui concerne notre premier objectif, qui consistait à montrer qu'il est possible de juger de la qualité d'un système orienté objet par simple analyse visuelle sans devoir se plonger dans l'analyse de valeurs multiples et de développer une méthode permettant cela, nous pouvons dire que nos buts ont été atteints, excepté la validation empirique de nos travaux. Nous n'avons donc pas de résultats proprement dit, mais nous avons mis au point une méthode et un guide permettant de donner des évaluations de certaines qualités de logiciels le plus objectivement possible. Nous avons par ailleurs évalué par cette méthode tout notre échantillon, et ces évaluations apparaissent sous forme de scores que nous présenterons dans les résultats de notre second objectif.

Notre deuxième axe de recherche concernant la complexité cyclomatique nous a donné des résultats intéressants. Nous avons évalué tout notre échantillon selon la méthode décrite à la section 3.5.1 (page 75). Nous avons donc d'abord donné un score entre 0 et 10 au couplage général des systèmes, à leur cohésion générale, à l'utilisation de l'héritage faite dans les systèmes, et ensuite nous avons évalué la complexité générale des systèmes en donnant aussi une évaluation sur une échelle de 0 à 10. La figure 4.1 représente notre tableau de scores constitué des évaluations de tous les systèmes qui constituent notre échantillon.

Un fois notre tableau de scores constitué, nous pouvions nous mettre à la recherche de corrélations entre les trois premières caractéristiques orientées objet observées de façon générale pour tout le système (couplage, cohésion et mécanisme d'héritage) et la complexité cyclomatique également observée pour tout le système à la fois. Comme nous sommes persuadés que plusieurs des trois premières caractéristiques influencent ensemble la complexité cyclomatique, nous avons recherché des liens entre une combinaison linéaire

1		DIT /10	CBO /10	LCOM5 /10	WMCC /10
2	Apache Ant 1.5	6	6	6	7
3	Art Of Illusion	4	2	4	4
4	Animal Shelter Manager	3	4	7	6
5	Axis	8	2	4	6
6	Azureus	9	8	7	9
7	Batik	7	7	7	8
8	Bouncy Castle Crypto APIs	8	9	5	8
9	Bots'n'Scouts	3	3	5	5
10	Castor	8	3	3	5
11	Cayenne Object-Relational Framework	7	4	3	6
12	Checkstyle	9	8	7	8
13	Columba	8	4	5	8
14	DbForms	8	2	6	6
15	Derby	8	5	3	6
16	Entagged - The Musical Box	7	8	7	9
17	FreeNet	8	3	2	5
18	Gantt Project	7	7	4	7
19	Groovy	7	6	2	6
20	Hibernate	7	5	4	6
21	HTML Parser	8	8	4	7
22	ICU	3	7	3	4
23	iText	6	4	3	4
24	JabRef	4	4	4	4
25	Jajuk	3	7	7	7
26	JasperReports	6	6	5	6
27	Jaxen	8	9	8	9
28	Jena	8	3	4	6
29	JFreeChart	6	6	6	6
30	JHylaFAX	5	8	6	8
31	Jmol	6	2	7	5
32	jExcelApi	8	3	6	7
33	JXTA	8	3	5	6
34	Liferay Portal	2	1	6	5
35	log4j	6	9	6	8
36	Logisim	3	4	7	6
37	Mantaray	6	4	5	6
38	MegaMek	7	3	3	4
39	Merchant Of Venice	6	2	7	7
40	PCGen	7	4	6	6
41	PDFBox	7	5	7	8
42	Phex	4	5	7	7
43	Pooka	6	3	6	5
44	ProGuard	6	9	4	8
45	Repast	7	5	4	6
46	RText	6	5	6	7
47	Sesame	7	8	7	7
48	Sphinx-4	6	5	5	6
49	SQuirreL SQL Client	6	3	7	7
50	Universe	7	6	6	8
51	Weka	8	1	6	5

FIG. 4.1 – *Tableau des scores donnés à chacun des logiciels de l'échantillon.*

des trois propriétés citées et la complexité cyclomatique du logiciel. La méthode la plus simple était de sommer les scores concernant le DIT, le CBO et celui de LCOM5 et de voir si cette somme était corrélée au score que nous donnions à la complexité cyclomatique (voir figure 4.1). Les premiers résultats sont intéressants. En effet, nous avons trouvé un coefficient de corrélation égal à **0.86677855472379** entre la somme des colonnes DIT + CBO + LCOM5 et la colonne WMCC. Cela signifie donc, que, selon notre méthode d'évaluation et selon la fiabilité qui lui est propre, la complexité cyclomatique générale d'un système est, dans une certaine mesure, corrélée à certaines des qualités orientées objet de ce système. Rappelons qu'ici nous évaluons surtout la maintenabilité des logiciels orientés objet grâce à notre méthode de visualisation, qui nous permet de dire si un système utilise bien ou non les avantages et les principes de l'Orienté Objet.

Les résultats obtenus ici sont intéressants mais il est évident qu'approfondir la recherche dans ce sens avec d'autres outils et d'autres données ne pourra que les enrichir.

4.2 Travaux futurs

4.2.1 Validation empirique des travaux et des résultats

Dans les travaux futurs qui pourraient enrichir cette recherche, nous recensons essentiellement un point très important. Nous avons décris dans ce travail une certaine méthodologie à suivre qui nous a mené à certains résultats. Ce qui nous manque à présent, c'est que ces résultats soient validés par d'autres, et que d'autres recherches suivant la même méthodologie soient pratiquées par d'autres chercheurs et experts afin de s'assurer qu'un manque d'objectivité n'a pas biaisé nos résultats. Nous désirons donc qu'une validation empirique vienne enrichir nos travaux. Nous avons obtenu certains résultats qui répondent à certaines de nos questions initiales, mais ils ne constituent qu'une validation mathématique d'une partie de nos travaux.

Nous avons mis sur pieds une méthode d'observation visuelle qui nous conduit à des indicateurs nous permettant d'évaluer certaines qualités orientées objet d'un système. Ces indicateurs doivent maintenant être validés par d'autres experts utilisant le même système de visualisation ou par d'autres techniques qui viendraient montrer en effet que ce que nous pouvons voir en une seule et brève observation est vérifiable par des valeurs ou autres indicateurs numériques par exemple. Cela nous permettrait d'être certain que ce système nous facilite, sans nous tromper, la tâche d'évaluation des qualités orientées objet d'un logiciel.

Le second objectif que nous avons réalisé est celui de montrer que la complexité cyclomatique observée au niveau du système entier permet de dire si un logiciel utilise bien l'Orienté Objet et ses principes (ce que nous pouvons dire grâce à nos indicateurs développés en premier lieu). Nous avons obtenu des résultats en suivant une méthode bien précise également. Ce sont ces résultats qui valident mathématiquement et en partie nos

hypothèses. Il serait maintenant intéressant que ces résultats soient validés ou invalidés par d'autres résultats que des techniques différentes de la nôtre peuvent fournir, ou encore par d'autres résultats que différents chercheurs auraient obtenus via notre méthode (avec un échantillon différent par exemple).

4.2.2 Nouvelle piste de recherche

En plus de la validation de nos résultats et de nos travaux, un autre point pourrait être intéressant à exploiter. Selon nous, notre théorie et notre méthode permettent de voir si un système profite bien ou non de l'Orienté Objet et de ses mécanismes. Nous savons que, dans la littérature, les *design patterns* sont reconnus comme de bonnes façons de programmer en Orienté Objet pour, notamment, faciliter la maintenance et la réutilisation. Il serait donc intéressant de pouvoir confronter ces méthodes avec notre technique d'évaluation et de voir si notre système permet de reconnaître ces types d'architectures ou, simplement, si notre système nous permet d'affirmer que le système est "bien programmé" quand celui-ci est un logiciel développé avec ce type de méthodes reconnues.

Ces différents axes de travaux futurs donneraient une plus grande valeur aux recherches actuelles et préciseraient certains points qui restent encore à l'état de suppositions actuellement.

4.3 Conclusion générale

En conclusion de ce mémoire, nous pouvons dire que les objectifs que nous nous étions fixés ont été presque tous atteints, exceptés la validation à plus grande échelle de nos résultats. Les recherches menées ici ouvrent la voie à de nouvelles possibilités et à l'exploitation de nouvelles pistes. Cette méthode d'observation, de visualisation et d'évaluation des qualités d'un logiciel orienté objet est inédite et, selon les premiers résultats obtenus, permet d'accélérer le processus d'amélioration d'un logiciel, vu que ce processus dépend de celui de l'évaluation. Si nous sommes capables de repérer les anomalies et les problèmes d'un logiciel très rapidement et précisément, il est certain que sa modification dans le but d'améliorer sa maintenabilité se fera plus rapidement et plus facilement car les problèmes auront été bien cernés préalablement. Nous pouvons dire que ce travail a fixé les bases d'utilisation de cette méthode et que maintenant il est impératif de voir son utilisation rendue plus populaire pour s'assurer de sa validité.

Nous pouvons enfin terminer en posant une question fondamentale autour de laquelle s'est orientée une partie du document: *"Est-il possible de voir un jour la complexité cyclomatique de MacCabe reconnue en tant qu'indicateur de bonne ou mauvaise utilisation du paradigme Orienté Objet?"*. Les premiers résultats obtenus et d'autres recherches dans ce domaine ouvrent de nouvelles pistes prometteuses mais il est évident que beaucoup reste à faire. Cette question et beaucoup d'autres nous montrent que le débat sur la qualité des

logiciels, orientés objet ou autres, est encore très ouvert et que beaucoup d'opinions se rencontrent. Ce qui nous laisse croire que nous ne sommes malheureusement pas encore à l'aube de l'élaboration et surtout de l'acceptation générale d'un indicateur simple et rapide de qualité d'un logiciel.

Bibliographie

[BB04] Linda BADRI and Mourad BADRI. A new class cohesion criterion: An empirical study on several systems. *Journal of Object Technology*, 3(4), April 2004.

[BBM96] Victor R. BASILI, Lionel C. BRIAND, and Walcelio L. MELO. A validation of Object-Oriented design metrics as quality indicators. *IEEE Transactions On Software Engineering*, 22(10):751–761, October 1996.

[BOE73] Barry W. BOEHM. Software and its impact: A quantitative assessment. *Datamation*, 19(5):48–59, May 1973.

[BOE78] Barry W. BOEHM. *Characteristics of Software Quality.* North Holland, 1978.

[CK94] Shyam R. CHIDAMBER and Chris F. KEMERER. A metrics suite for Object Oriented design. *IEEE Transactions On Software Engineering*, 20(6):476 – 493, June 1994.

[eA95] Fernando Brito e ABREU. The MOOD metrics set. *ECOOP '95 Workshop on Metrics*, 1995.

[EL96] Karin ERNI and Claus LEWERENTZ. Applying design-metrics to Object-Oriented frameworks. *Proceedings of METRICS 1996*, 1996.

[FOW] Martin FOWLER. Refactoring home page. www.refactoring.com.

[FP97] Norman E. FENTON and Shari Lawrence PFLEEGER. *Software Metrics : A Rigorous and Practical Approach.* International Thompson Computer Press, 2nd edition, 1997.

[GHJV94] Erich GAMMA, Richard HELM, Ralph JOHNSON, and John VLISSIDES. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1st edition, 1994.

[GUÉ03] Yann-Gaël GUÉHÉNEUC. *Un cadre pour la traçabilité des motifs de conception.* PhD thesis, École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, June 2003.

[GUÉ05] Y-G. GUÉHÉNEUC. Ptidej: Promoting patterns with patterns. *ECOOP 2005*, 2005.

[HALS06] Naji HABRA, Alain ABRAN, Miguel LOPEZ, and Asma SELLAMI. A framework for software measurement design. *Research Report RR-33/06, Institut d'Informatique, FUNDP Namur*, 2006.

[HC98] Rachel HARRISON and Steve J. COUNSELL. An evaluation of the MOOD set of Object-Oriented software metrics. *IEEE Transactions On Software Engineering*, 24(6):491–496, June 1998.

[HD01] Matthew HOLLY and Karel DRIESEN. Visualizing indirect branch hot spots in Object-Oriented programs. *Workshop on Software Visualization, OOPSLA 2001*, 2001.

[HS96] Brian HENDERSON-SELLERS. *Object-Oriented Metrics Measures of Complexity*. Prentice-Hall, 1996.

[HVW05] Danny HOLTEN, Roel VLIEGEN, and Jarke J. VAN WIJK. Visual realism for the visualization of software metrics. 2005.

[LD01] Michele LANZA and Stéphane DUCASSE. The class blueprint, a visualization of the internal structure of classes. *Workshop on Software Visualization, OOPSLA 2001*, 2001.

[LSP05] G. LANGELIER, H. SAHRAOUI, and P. POULIN. Visualization-based analysis of quality for largescale software systems. *IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[LY96] Tim LINDHOLM and Frank YELLIN. *The Java Virtual Machine Specification*. Sun Microsystems, Inc., 2nd edition, 1996.

[MAR95] Robert MARTIN. OO design quality metrics, an analysis of dependencies. *Report on Object Analysis & Design*, 2(3), 1995.

[MAR01] Stuart MARSHALL. Using and visualizing reusable code. *Workshop on Software Visualization, OOPSLA 2001*, 2001.

[MAR03] Avichay MARTCIANO. Measuring the quality of programs. Technion - Israeli Institute of Technology, Computer Science faculty, 2003.

[McC76] Thomas J. McCABE. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[MG05] Naouel MOHA and Yann-Gaël GUÉHÉNEUC. On the automatic detection and correction of software architectural defects in object-oriented designs. *ECOOP 2005*, 2005.

[MHG06] Naouel MOHA, Duc-Loc HUYNH, and Yann-Gaël GUÉHÉNEUC. Une taxonomie et un métamodèle pour la détection des défauts de conception. *LMO 2006*, 2006.

[MÄN03] Mika MÄNTYLÄ. Bad smells in software - a taxonomy and a empirical study. Master's thesis, Helsinky University of Technology, May 2003.

[SLPH05] Grégory SERONT, Miguel LOPEZ, Valérie PAULUS, and Naji HABRA. On the relationship between cyclomatic complexity and the degree of object orientation. *QAOOSE 2005*, 2005.

[SSL01] Frank SIMON, Frank STEINBRUCKNER, and Claus LEWERENTZ. Metrics based refactoring. *IEEE CSMR 2001*, pages 30–38, 2001.

[STR00] Alfred STROHMEIER. Cycle de vie du logiciel. Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, 2000.

[TK03] Ladan TAHVILDARI and Kostas KONTOGIANIS. A metric-based approach to enhance design quality through meta-pattern transformations. *IEEE CSMR 2003*, 2003.

[ZAI04] Farouk ZAIDI. POM, un outil pour le calcul des métriques de qualité. Projet de fin d'études, Université de Technologie de Belfort-Montbéliard, 2004.

Troisième partie

Annexe

Composition de l'échantillon

Dans cette annexe, nous décrirons la composition de notre échantillon constitué de 50 systèmes codés en Java. Pour chaque système, nous donnerons un très brève description de ses fonctionnalités ainsi que son nombre de classes, ce qui permet d'estimer sa taille. Tous les systèmes faisant partie de cet échantillon proviennent de la communauté Open Source. Les différents codes ont été téléchargés via le portail *SourceForge.net*.

1. **Apache Ant 1.5 :**
   - **Description:** "Apache Ant" est un outil écrit en Java. C'est principalement un outil de compilation Java (automatisation de la construction de projet Java), mais il vise plus généralement le développement d'un logiciel d'automatisation des opérations répétitives tout au long du cycle de développement d'un système.
   - **Nombre de classes:** 401
2. **Art of Illusion :**
   - **Description:** "Art of Illusion" est un logiciel multi-plateforme pour s'initier à la 3D. Il est développé entièrement en Java. Il permet la création et le dessin en trois dimensions ainsi que l'édition procédurale de textures.
   - **Nombre de classes:** 868
3. **Animal Shelter Manager :**
   - **Description:** "Animal Shelter Manager" est un logiciel pour la gestion des animaux au sein d'un centre d'accueil ou de soins. Il permet de se libérer des tâches administratives engendrées pour un refuge pour animaux ou encore pour une clinique vétérinaire.
   - **Nombre de classes:** 960
4. **Axis :**
   - **Description:** "Axis (Apache eXtensible Interaction System)" est une nouvelle implémentation de la spécification SOAP (Simple Object Access Protocol). SOAP est, entre autres,un protocole de communication basé sur XML pour permettre aux applications de s'échanger des informations via HTTP. "Axis" est à la fois un environnement d'hébergement de services Web, et un *toolkit* complet de développement pour la création de services et l'accès à des services tiers.
   - **Nombre de classes:** 756

5. **Azureus :**
   - **Description:** "Azureus" est un logiciel de partage de ficher *peer to peer*. C'est un client Bittorrent Open Source codé en Java.
   - **Nombre de classes:** 1580
6. **Batik :**
   - **Description:** "Batik" est un jeu de modules de base qui peuvent être utilisés ensemble ou séparément pour supporter des solutions SVG (Scalable Vector Graphics). "Batik" est extrêmement extensible et est constitué de plusieurs composants qui permettent de visualiser, de produire des images à partir de sources SVG, et d'exporter des affichages en SVG.
   - **Nombre de classes:** 1558
7. **Bouncy Castle Crypto APIs :**
   - **Description:** "Bouncy Castle Crypto APIs" est une librairie Java implémentant divers algorithmes cryptographiques.
   - **Nombre de classes:** 672
8. **Bots'n'Scouts :**
   - **Description:** "Bots'n'Scouts" est un jeu de stratégie multijoueurs programmé en Java. Le but est de créer un robot pour le faire participer à une course.
   - **Nombre de classes:** 260
9. **Castor :**
   - **Description:** "Castor" est un *framework* de liaison de données pour Java. Il constitue un chemin simplifié entre les objets Java, les documents XML et tables de relations dans les bases de données.
   - **Nombre de classes:** 955
10. **Cayenne :**
    - **Description:** "Cayenne" est un ensemble d'outils écrits en Java permettant de travailler avec des bases de données relationnelles d'une façon "Orienté-Objet".
    - **Nombre de classes:** 1836
11. **Checkstyle :**
    - **Description:** "Checkstyle" est un outil qui propose de puissantes fonctionnalités pour appliquer des contrôles sur le respect de règles de codification.
    - **Nombre de classes:** 767
12. **Columba :**
    - **Description:** "Columba" est un client e-mail écrit en Java. Il comprend une interface graphique et est aussi un outil puissant de gestion de courrier.
    - **Nombre de classes:** 1191
13. **DbForms :**
    - **Description:** "DbForms" est un outil de développement Java qui permet d'élaborer des formulaires de saisie et d'accès aux bases de données.

- **Nombre de classes:** 242

14. **Derby :**
    - **Description:** "Derby" (anciennement "Cloudscape" de IBM) est une base de données relationnelle entièrement en Java.
    - **Nombre de classes:** 1183
15. **Entagged - The Musical Box :**
    - **Description:** "Entagged - The Musical Box" est un éditeur d'informations musicales contenues dans les fichiers audionumériques (ex: artiste, album, etc). Les formats supportés sont: MP3, OGG, FLAC, MPC, APE et WMA.
    - **Nombre de classes:** 443
16. **Freenet :**
    - **Description:** "Freenet" est un réseau informatique et décentralisé bâti au dessus d'Internet visant à permettre une liberté d'expression et d'information totale profitant de la sécurité et de l'anonymat. Le logiciel analysé ici est le logiciel client permettant de communiquer selon le protocole "Freenet".
    - **Nombre de classes:** 1400
17. **Gantt Project :**
    - **Description:** "Gantt Project" est une application développée en Java, permettant de réaliser des diagrammes de Gantt documentés. Ce programme possède des capacités d'exportation en une multitude de formats.
    - **Nombre de classes:** 1070
18. **Groovy :**
    - **Description:** "Groovy" et un langage de script pour Java qui apporte directement des fonctionnalités de langages tels que Pyhton, Ruby ou Smalltalk. "Groovy" peut aussi être utilisé comme un compilateur alternatif pour générer du *bytecode* Java standard.
    - **Nombre de classes:** 869
19. **Hibernate :**
    - **Description:** "Hibernate" est un *framework* Java de persistance qui permet de faire correspondre des tables de base de données relationnelles avec des objets Java simples. "Hibernate" permet aussi de manipuler des fichiers XML pour pouvoir les mettre en rapport avec des bases de données relationnelles.
    - **Nombre de classes:** 955
20. **HTMLParser :**
    - **Description:** "HTMLParser" est un *parser* en temps réel très rapide, caractérisé par un design très simple. Il est utilisé pour la transformation et l'extraction de pages HTML.
    - **Nombre de classes:** 152

21. **ICU :**
    - **Description:** "ICU (International Components for Unicode)" offre des services robustes dotés de toutes les fonctionnalités pour analyser, comparer et transformer les chaînes de textes Unicode.
    - **Nombre de classes:** 511
22. **iText :**
    - **Description:** "iText" est une API permettant à partir d'un programme Java de générer des documents PDF et HTML.
    - **Nombre de classes:** 573
23. **JabRef :**
    - **Description:** "JabRef" est un outil graphique de gestion de base de données bibliographique BibTeX, format standard LaTeXdes références bibliographiques.
    - **Nombre de classes:** 1315
24. **Jajuk :**
    - **Description:** "Jajuk" est un Jukebox MP3/OGG/WAV/AU/AIFF/SPEEX écrit en Java. Ce n'est pas un simple lecteur mais aussi un système de gestion de collection musicale.
    - **Nombre de classes:** 379
25. **JasperReports :**
    - **Description:** "JasperReports" est un ensemble de librairies Java dédié à la génération de rapport sous divers formats. La bibliothèque de "JasperReports" est capable de fournir du contenu riche à l'écran ou dans des fichiers PDF, HTML, XLS, CSV et XML.
    - **Nombre de classes:** 583
26. **Jaxen :**
    - **Description:** "Jaxen" et un moteur *Xpath* écrit en Java qui permet de retrouver des informations grâce à *Xpath* dans un document XML de type *dom4j* ou *Jdom*. C'est un projet Open Source qui a été intégré dans *dom4j* pour permettre le support de *Xpath* dans ce *framework*.
    - **Nombre de classes:** 191
27. **Jena :**
    - **Description:** "Jena" est une bibliothèque de classes Java qui facilite le développement d'applications pour le Web sémantique. "Jena" facilite la manipulation de déclarations RDF, la lecture et écriture RDF/XML, et le stockage en mémoire ou sur disque de connaissances RDF. "Jena" est aussi utilisé pour interroger une base RDF, et pour la gestion d'ontologies.
    - **Nombre de classes:** 1587
28. **JFreeChart :**
    - **Description:** "JFreeChart" est une bibliothèque permettant la génération de graphiques de type histogramme, courbe, etc... Cette bibliothèque offre tout le

nécessaire pour représenter des données chiffrées en graphiques pour l'utilisateur, dans une application, dans une image ou vers une imprimante.
   - **Nombre de classes:** 485

29. **JHylaFAX :**
   - **Description:** "JHylaFAX" est un programme client multi-plateformes pour "HylaFax". Il est capable d'envoyer des faxs, d'afficher le statut du serveur et de recevoir des faxs.
   - **Nombre de classes:** 581
30. **Jmol :**
   - **Description:** "Jmol" est un outil multi-plateformes de visualisation animée en trois dimensions de molécules gratuit, pour les étudiants, les éducateurs et les chercheurs en chimie et en biochimie.
   - **Nombre de classes:** 813
31. **JExcelApi :**
   - **Description:** "JExcelApi" est une bibliothèque Java qui permet de lire et d'écrire des documents Microsoft Excel.
   - **Nombre de classes:** 477
32. **JXTA :**
   - **Description:** La technologie "JXTA", initiée par Sun, est un ensemble de protocoles ouverts concernant les liaisons *peer-to-peer*, permettant de connecter et de faire collaborer n'importe quel matériel connecté au réseau.
   - **Nombre de classes:** 787
33. **Liferay Portal :**
   - **Description:** "Liferay Portal" est une solution Open Source J2EE. Ce portail est compatible avec divers serveurs d'application. La force de ce portail est de proposer un grand nombre de portlets: webmail, bibliothèque de documents et d'images, forums, agenda, ...
   - **Nombre de classes:** 2160
34. **log4j :**
   - **Description:** "log4j" est un outil de log développé par Apache. C'est un jeu d'API qui permet aux développeurs de générer des logs depuis leur code et de configurer ensuite le niveau, la sortie et le formatage des logs à l'aide de fichiers de configuration externes à l'application.
   - **Nombre de classes:** 244
35. **Logisim :**
   - **Description:** "Logisim" est un outil éducatif pour dessiner et simuler des circuits électroniques logiques. Il est notamment composé d'une interface simple d'accès.
   - **Nombre de classes:** 899

36. **Mantaray :**
    - **Description:** "Mantaray" est un middleware 100% Java de communication et de messagerie complétement distribué pair-à-pair qui fontionne sans serveur. Il offre la garantie de livraison, de sécurité et de transaction.
    - **Nombre de classes:** 453
37. **MegaMek :**
    - **Description:** "MegaMek" est un jeu de plateau écrit en Java. Il est un clone non-officiel du jeu de plateau devenu classique: BattleTech. Il se joue en réseau à 2 joueurs ou plus, dont l'un peut être une IA pour les partie *offline*.
    - **Nombre de classes:** 627
38. **Merchant Of Venice :**
    - **Description:** "Merchant Of Venice" est un programme d'échange d'actions boursières qui permet aussi la gestion de portefeuille, les analyses techniques et d'autres fonctions boursières habituelles. "Merchant Of Venice" offre une interface graphique utilisateur avec de l'aide en ligne ainsi qu'une documentation complète.
    - **Nombre de classes:** 1340
39. **PCGen :**
    - **Description:** "PCGen" est un logiciel de gestion de personnages de jeux de rôle version papier.
    - **Nombre de classes:** 1762
40. **PDFBox :**
    - **Description:** "PDFBox" est une API permettant la création et la manipulation de fichiers PDF. "PDFBox" fournit aussi un ensemble d'outils utilisables en ligne de commande.
    - **Nombre de classes:** 373
41. **Phex :**
    - **Description:** "Phex" est un client *peer-to-peer* d'échange de fichier qui est basé sur le réseau Gnutella. "Phex" propose tous les services classiques propres à Gnutella.
    - **Nombre de classes:** 1125
42. **Pooka :**
    - **Description:** "Pooka" est un client e-mail écrit en Java, utilisant Swing et JavaMail. Il supporte les protocoles IMAP et POP3. Il offre aussi un support pour le cryptage utilisant PGP et S/MIME.
    - **Nombre de classes:** 1329
43. **ProGuard :**
    - **Description:** "ProGuard" est un obfuscateur (compresseur de code) de code Java, développé en Java et Open Source. Il permet de brouiller et d'optimiser un code Java.

- **Nombre de classes:** 318

44. **Repast :**
    - **Description:** "Repast" est un *framework* pour créer des simulations orientées agents. Il fournit une librairie de classes pour créer, lancer, afficher et récolter des données d'une simulation orientée agent.
    - **Nombre de classes:** 739
45. **RText :**
    - **Description:** "RText" est un éditeur de texte modifiable pour programmeur. Il offre tous les services d'un éditeur de code habituel comme la reconnaissance des mots clés, la possibilité d'éditer plusieurs documents à la fois,...
    - **Nombre de classes:** 546
46. **Sesame :**
    - **Description:** "Sesame" est un *framework* Java permettant de stocker, interroger et manipuler des informations RDF et RDF Schéma. Il supporte plusieurs supports de stockage, les langages de requêtes SeRQL, RQL et RDQL, permet de traiter du RDF/XML, des N-triplets, Turtle et N3. Il fournit également des modules de gestion d'ontologies et bien d'autres fonctionnalités.
    - **Nombre de classes:** 503
47. **Sphinx-4 :**
    - **Description:** "Sphinx-4" est un système de reconnaissance de la parole. C'est un système très flexible capable d'accomplir de multiples tâches différentes de reconnaissance de la parole.
    - **Nombre de classes:** 396
48. **SQuirreL SQL Client :**
    - **Description:** "SQuirreL SQL Client" est un outil d'interrogation SQL munit d'une interface graphique.
    - **Nombre de classes:** 729
49. **Universe :**
    - **Description:** "Universe" est un jeu multijoueurs au tour par tour. C'est un jeu de stratégie spatiale similaire à "Master Of Orion".
    - **Nombre de classes:** 656
50. **Weka :**
    - **Description:** "Weka" est une collection d'algorithmes d'apprentissage (*Machine Learning*) pour résoudre des problèmes réels de fouille de données (*Data Mining*). Les algorithmes peuvent aussi être appliqués directement sur un ensemble de données ou être appelés à partir d'un code Java.
    - **Nombre de classes:** 1382