Université de Montréal

Détection visuelle d'anomalies de conception dans les programmes orientés objets

par
Karim Dhambri

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

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

décembre, 2007

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Détection visuelle d'anomalies de conception dans les programmes orientés objets

présenté par:

Karim Dhambri

a été évalué par un jury composé des personnes suivantes:

Yann-Gaël Guéhéneuc
président-rapporteur

Houari A. Sahraoui
directeur de recherche

Pierre Poulin
codirecteur

Marc Feeley
membre du jury

Mémoire accepté le

RÉSUMÉ

De nos jours, les logiciels doivent être flexibles pour pouvoir accommoder d'éventuels changements. Les anomalies de conception introduites durant l'évolution du logiciel causent souvent des difficultés de maintenance et un manque de flexibilité aux changements futurs. Dû à la nature des connaissances requises, un important sous-ensemble d'anomalies de conception est difficile à détecter de façon automatique. Malheureusement, la détection manuelle est coûteuse en temps et ressources. Dans le cadre de cette maîtrise, nous proposons une approche semi-automatique de détection d'anomalies de conception basée sur la visualisation. Nous introduisons un principe de détection général et l'illustrons sur un ensemble d'anomalies avec des exemples concrets. Nous discutons finalement des avantages et des limitations de notre stratégie à l'aide de deux études de cas.

Mots clés: détection, anomalies de conception, visualisation, métriques.

ABSTRACT

Nowadays, software must be flexible to accommodate future changes. Design anomalies, introduced during software evolution, are frequent causes of low maintainability and low flexibility to future changes. Because of the required knowledge, an important subset of design anomalies are difficult to detect automatically. Unfortunately, manual detection through code inspection is too time- and resource-consuming. In the context of this master degree, we propose a visualization-based approach to semi-automatic detection of design anomalies. We introduce a general detection principle and illustrate it on a set of anomalies with concrete examples. We finally discuss the advantages and limitations of our strategy through two case studies.

Keywords: detection, design anomalies, visualization, metrics.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES

ATFD	Access to Foreign Data
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
LCOM	Lack of Cohesion in Methods
LOC	Lines Of Code
NCC	Number of Client Classes
NCP	Number of Client Packages
NMD	Number of Methods Declared
NOCCP	Number Of Client Classes of a Package
NPDM	Number of Public Declared Methods
NUC	Number of Used Classes
OO	Orienté Objet
PADL	Pattern and Abstract-level Description Language
PC	Package Cohesion
POM	Primitives, Operators, Metrics
RCU	Relative Class Utility
TCC	Tight Class Cohesion
VERSO	Visualization for Evaluation and Re-engineering of object-oriented SOftware
WMC	Weighted Methods per Class

À ma famille

REMERCIEMENTS

J'aimerais tout d'abord remercier mes directeurs Houari Sahraoui et Pierre Poulin qui m'ont soutenu et porté conseil tout au long de mes études. Ils ont également bien voulu relire ce mémoire durant les vacances de Noël. Je remercie Yann-Gaël Guéhéneuc qui a toujours été disponible pour répondre à mes questions.

Je voudrais aussi remercier mes collègues Guillaume Langelier et Salima Hassaine avec qui j'ai travaillé en étroite collaboration sur le projet VERSO, ainsi que Naouel Moha avec qui j'ai échangé des résultats sur la détection d'anomalies. Je remercie également Duc-Loc Huynh pour mettre de l'ambiance au laboratoire GEODES et Stéphane Vaucher pour toutes les conversations que nous avons eues autour d'un bon café. Je tiens aussi à remercier David St-Hilaire et David Pouliot, mes partenaires de jeux vidéo à l'heure du diner.

Finalement, je remercie les membres de ma famille, Maude Dhambri, Nicole Choinière, Rachid Dhambri, Mariane Cantin et Fouzia Dhambri, ainsi que mes amis de longue date, Sébastien Lacroix, Marc-André Sawyer et Jean-François Soucy, pour leurs nombreux encouragements depuis le début de mes études universitaires.

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Les logiciels orientés objet (OO) sont en constante évolution. Dans plusieurs domaines, les programmes requièrent des changements fréquents, et ce à tout niveau de considération. Un logiciel ne peut plus simplement répondre aux spécifications fonctionnelles ; il doit être flexible aux changements futurs. Les anomalies de conception (*i.e.*, des déviations des bonnes pratiques de conception de logiciels) introduites durant les processus de développement et de maintenance peuvent compromettre la facilité d'évolution d'un logiciel. Selon Fenton et Pfleeger [10], les anomalies de conception sont rarement la cause directe d'une défaillance, mais elles le sont souvent indirectement. La détection et la correction de ces anomalies sont donc une contribution concrète à l'amélioration de la qualité du logiciel.

1.2 Motivation

Cependant, la détection d'anomalies de conception n'est pas triviale [25]. La détection manuelle est coûteuse en temps et en ressources, alors que la détection automatique génère beaucoup de faux positifs. Ces faux positifs sont dus à la nature des connaissances impliquées dans la détection [27]. De plus, il existe des difficultés additionnelles inhérentes à la détection d'anomalies, telles que la dépendance au contexte, la taille de l'espace de recherche, l'ambiguïté des définitions, ainsi que le problème bien connu de définition de valeurs seuil lorsque la détection utilise des aspects quantitatifs (métriques) [25]. Le reste de cette section relève quelques questions ouvertes liées à la détection d'anomalies de conception.

1.2.1 Décider de la pertinence d'un candidat d'anomalie

Contrairement aux défaillances, il n'existe pas de consensus à savoir si un design donné va à l'encontre des heuristiques de qualité. Il y a une différence entre détecter des symptômes et affirmer hors de tout doute qu'une situation détectée est vraiment une anomalie de conception. L'intervention humaine est nécessaire pour comprendre la situation détectée et décider si une correction s'impose.

1.2.2 Traiter une longue liste de candidats d'anomalies

Détecter un grand nombre d'occurrences d'anomalies dans un système n'est pas toujours utile. La présence de faux positifs peut affecter la coopération des équipes de développement. Si l'on signale régulièrement à ces équipes la présence d'anomalies alors qu'il s'agit en fait de faux positifs, elles peuvent mettre en doute la pertinence de ces alertes et rejeter systématiquement tout commentaire concernant la qualité du logiciel par la suite. De plus, comprendre et corriger les occurrences d'anomalies de conception est un processus long, coûteux et pas toujours profitable.

1.2.3 Déterminer les frontières

Il y a un consensus général sur les manifestations extrêmes d'anomalies de conception. Par exemple, considérons un programme OO de plusieurs centaines de classes, dont l'une d'elles implémente tout le comportement et toutes les autres sont seulement des classes avec des attributs et des méthodes pour les accéder. Il s'agit sans aucun doute d'une occurrence de *blob* (voir la Section 3.1). Malheureusement, dans des systèmes réels, on peut trouver plusieurs classes de grande taille qui utilisent à la fois des classes de données ainsi que des classes régulières. Décider lesquelles sont des occurrences de *blob* dépend beaucoup de l'interprétation de l'analyste.

1.2.4 Définir les valeurs seuil

La détection du *blob* requiert de l'information quantitative comme la taille de la classe (une *grande* classe). Bien que l'on puisse mesurer la taille d'une classe (*e.g.*, le nombre de lignes de code), il n'est pas trivial de définir une valeur seuil appropriée. Une classe peut être considérée grande dans un programme donné, mais moyenne dans un autre.

1.2.5 Tenir compte du contexte

Une violation flagrante de principe de conception peut être considérée acceptable dans certains contextes. Par exemple, une classe *Log* responsable de maintenir le registre des événements dans un système, utilisée par un grand nombre de classes, est une pratique courante et acceptable. Cependant, par définition, elle pourrait être considérée comme un cas de *shotgun surgery* (voir section 3.1).

1.2.6 Extraire de l'information sémantique

Selon la définition de certaines anomalies, l'information sémantique (*i.e.*, le rôle fonctionnel joué par un artefact de logiciel) est cruciale à la détection. Pour l'anomalie décomposition fonctionnelle (voir la Section 3.1), une des conditions clé est qu'une classe implémente une fonction unique dans le système. Pour la détection automatique, on approxime cette information en utilisant la structure du programme ou en recherchant des mots clé dans les noms et les commentaires.

1.3 Contributions

Dans le cadre de cette maîtrise, nous proposons une approche semi-automatique de détection d'anomalies de conception basée sur la visualisation de logiciels. Notre outil de visualisation VERSO [22] présente quatre catégories d'informations à l'analyste : quantitative, architecturale, relationnelle et sémantique. Nous modélisons les anomalies de conception sous forme de scénarios, dans lesquels les classes jouent des

rôles primaires et secondaires. Ce modèle contribue à réduire l'espace de recherche pour la détection visuelle.

Notre approche est complémentaire à la détection automatique. En effet, nous nous concentrons précisément sur les anomalies qui sont difficiles à détecter de façon automatique. Nous utilisons des stratégies de détection qui combinent des actions automatisées avec des actions exécutées par l'usager. Le jugement de l'utilisateur est sollicité lorsque les décisions automatiques sont difficiles à prendre. De telles décisions sont liées au contexte de l'application, au choix de l'architecture de bas niveau, à la variabilité des occurrences d'anomalies et aux valeurs seuil des métriques.

Nous résumons les contributions de cette maîtrise comme suit :

- une approche semi-automatique de détection d'anomalies de conception basée sur la visualisation ;
- le développement de stratégies de détection pour six anomalies documentées dans la littérature ;
- le développement d'extensions à un outil de visualisation afin de supporter notre approche de détection.

En terminant, nous avons eu l'occasion de publier les résultats de nos travaux dans deux articles. Nous avons d'abord proposé une approche basée sur des taxonomies de tâches d'analyse de haut niveau, de tâches interactives et de règles de perception pour concevoir un assistant à la détection visuelle [18]. Ensuite, nous avons résumé la majeure partie du travail présenté dans ce mémoire, c'est à dire les enjeux de la détection visuelle, la représentation d'information, les stratégies de détection avec des exemples, ainsi qu'une étude de cas dans [7].

1.4 Structure du mémoire

Le reste du mémoire est divisé comme suit. Le Chapitre 2 présente un état de l'art sur les approches de détection et les systèmes de visualisation de logiciels.

Des problématiques liées à la détection d'anomalies de conception sont discutées au Chapitre 3, ainsi qu'une description des types d'informations nécessaires à la détection. Ce chapitre introduit également les anomalies de conception qui seront utilisées comme exemples dans le reste de ce mémoire. Le Chapitre 4 décrit de quelle façon nous avons étendu l'outil de visualisation VERSO pour permettre les tâches de détection et la façon dont nous représentons les types d'informations présentés au Chapitre 3. Notre principe général de détection, ainsi que des stratégies de détection spécifiques à chacune des anomalies introduites au Chapitre 3 sont présentés au Chapitre 5. Un exemple concret illustre chaque anomalie détectée. Le Chapitre 6 explore les avantages et les limitations de notre approche à l'aide des résultats de deux études de cas. Finalement, nous concluons la présentation de notre travail au Chapitre 7.

CHAPITRE 2

ÉTAT DE L'ART

Le travail réalisé dans le cadre de cette maîtrise puise ses sources dans plusieurs domaines d'étude du logiciel. Les définitions d'anomalies que nous utilisons dans ce travail proviennent de plusieurs recueils de définitions d'anomalies de conception dans les logiciels OO. Un bon nombre de travaux ont déjà été réalisés sur la détection d'anomalies de conception. Les types d'anomalies détectées varient du niveau système dans son ensemble jusqu'au niveau des classes et des méthodes. Notre travail combine la détection d'anomalies avec la visualisation de logiciels. Les approches de visualisation existantes offrent chacune certaines vues du logiciel, à différentes échelles. Ce chapitre présente les travaux pertinents à notre travail en ce qui a trait à la définition d'anomalies de conception, la détection d'anomalies et la visualisation de logiciels.

2.1 Définition d'anomalies de conception

La détection d'anomalies de conception dans des logiciels OO est un sujet d'intérêt depuis la dernière décennie dans la communauté du logiciel. Brown *et al.* [3] ont introduit le concept d'anti-patron de conception. Les anti-patrons sont la contre partie des patrons de conception [13]. Si un patron de conception est une bonne solution à un problème récurrent, un anti-patron est une mauvaise solution récurrente à des problèmes de conception. Les anti-patrons présentés par Brown *et al.* sont définis à plusieurs niveaux : classe, micro-architecture, *framework*, application, système, entreprise et industriel. Le spectre d'anti-patrons couvert est très large ; il passe du code spaghetti jusqu'aux problèmes de gestion de projets. Dans le cadre de notre travail, nous nous intéressons essentiellement aux niveaux classe et micro-architecture.

Quelques années plus tard, Fowler *et al.* [12] proposèrent la notion d'odeurs de code (*code smells*). Une odeur de code n'est pas nécessairement un défaut de conception, mais plutôt un symptôme qui indique qu'il pourrait y avoir un problème nécessitant une refactorisation. Les odeurs de code décrites dans cet ouvrage traitent d'héritage, d'appels entre les classes et de la structure interne des classes.

Moha *et al.* [29] présentent une taxonomie d'anti-patrons et d'odeurs de code. Cette classification, présentée à la figure 2.1, fait ressortir les similarités et les différences parmi ces défauts de conception. Cette taxonomie vise à faciliter l'identification et limiter les problèmes d'interprétation des anti-patrons et des odeurs de code.

D'autres travaux se sont intéressés à l'étude d'heuristiques de qualité de conception OO. Riel [31] présente une soixantaine de règles de bonne conception OO, sans égard au langage de programmation utilisé. L'auteur ne prétend pas que ces règles doivent être suivies à tout prix, mais plutôt qu'elles devraient servir à guider les choix de conception d'un programme. Martin [26] propose des principes d'architecture de package dans les logiciels OO. Par exemple, selon son *principe de réutilisation commune*, des classes qui ne sont pas réutilisées ensemble ne devraient pas être groupées ensemble. Une conception de logiciel allant à l'encontre des heuristiques et principes décrits dans ces ouvrages peuvent être considérés comme des anomalies de conception.

2.2 Détection d'anomalies

Malgré l'importance de la détection d'anomalies de conception pour l'amélioration de la qualité du logiciel, peu d'approches on été proposées jusqu'à maintenant.

Alikacem et Sahraoui [1] présentent une approche de détection de violations des règles de qualité dans les programmes OO. Cette approche modélise ces règles par des systèmes à base de règles floues agissant sur le code source. Les auteurs

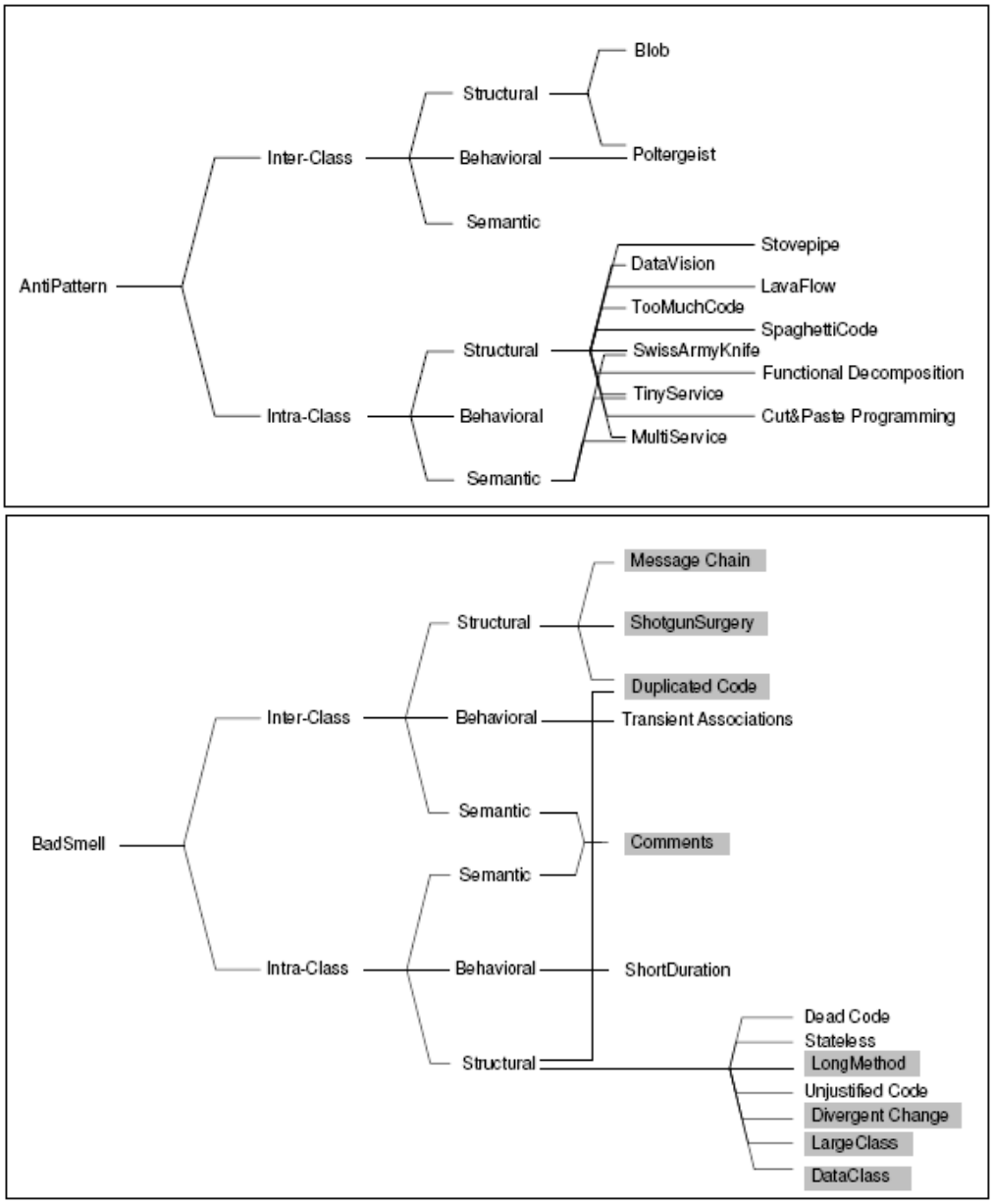


FIG. 2.1 – Taxonomie d'anti-patrons et d'odeurs de code par Moha *et al.* [29]. Les odeurs de codes grisées sont définies par Fowler *et al.* [12].

proposent également un langage de description des règles de qualité composé d'un nombre restreint de constructions syntaxiques simples afin qu'il puisse être utilisé par des non programmeurs. La figure 2.2 présente un système de règles floues exprimant la détection du *blob*.

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

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

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

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

FIG. 2.2 – Système de règles floues exprimant la détection du *blob* par Alikacem et Sahraoui [1].

Moha *et al.* [27] proposent un langage pour spécifier formellement les règles de détection d'anomalies. La spécification des règles de détection pour le *blob* est présentée à la figure 2.3. Ces spécifications sont ensuite utilisées pour générer automatiquement des algorithmes de détection qui servent à détecter automatiquement l'anomalie spécifiée.

À propos du processus de détection, Ciupke [5] propose une technique d'analyse de code source, en spécifiant les problèmes de conception sous forme de requêtes, et de localisation d'occurrences de ces problèmes dans un modèle dérivé du code source. La majorité des anomalies détectées sont relativement simples, *i.e.*, de simples conditions avec valeurs seuil fixes telles que "la profondeur de l'arbre d'héritage ne doit pas excéder six niveaux". Les problèmes complexes, comme la détection d'anti-patrons, ne sont pas traités dans ce travail.

Marinescu [25] propose une approche complète pour la détection d'anti-patrons. Marinescu utilise des stratégies de détection basées sur les métriques. Une stratégie de détection est une composition de règles. Chaque règle correspond à filtrer les classes du système pour en trouver un sous-ensemble répondant à un certain critère.

```
1    RULE_CARD: BlobCard {
2
3    RULE: Blob {ASSOC: associated FROM: ControllerClass ONE
4                                   TO: DataClass MANY};
5    RULE: ControllerClass
6          {INTER LargeClassLowCohesion ControllerClassName};
7
8    RULE: LargeClassLowCohesion
9          {INTER LargeClass ClassLowCohesion};
10
11   RULE: LargeClass {(METRIC: NM + NA, VERY_HIGH)};
12
13   RULE: ClassLowCohesion {(METRIC: LCOM5, VERY_HIGH)} ;
14
15   RULE: ClassControllerName {UNION
16          (SEMANTIC: CLASSNAME, {System, Manager, Controller})
17          (SEMANTIC: METHODNAME, {Process, Control, Command})};
18
19   RULE: DataClass
20         {INTER (STRUCT: METHOD, Accessor) (METRIC: LCOM5, HIGH)};
21   };
```

FIG. 2.3 – Carte de règles de détection du *blob* par Moha *et al.* [27].

Les critères peuvent être absolus ou relatifs. Par exemple, il est possible de spécifier comme règle “les classes ayant une valeur supérieure à six pour la métrique WMC (*Weighted Methods per Class*)”, ou bien “10% des classes ayant les plus grandes valeurs de WMC”. La stratégie de détection sert donc à former un sous-ensemble de classes correspondant à l'intersection de tous les sous-ensembles spécifiés par les règles de détection. La figure 2.4 présente la spécification proposée par Marinescu pour détecter les occurrences de *blob* dans un système. Cette stratégie de détection s'interprète de la manière suivante : le sous-ensemble des classes du système S pour lesquelles WMC fait partie du quartile supérieur, ATFD (*Access to Foreign Data*) est supérieur à 1 et TCC (*Tight Class Cohesion*) fait partie du quartile inférieur.

$$GodClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (WMC(C), TopValues(25\%)) \wedge (ATFD(C), HigherThan(1)) \wedge (TCC, BottomValues(25\%)) \end{array} \right.$$

FIG. 2.4 – Stratégie de détection pour le *blob* spécifiée par Marinescu [25].

Marinescu a développé de telles stratégies de détection pour dix anomalies de conception OO. Cependant, les taux de précision de la détection varient beaucoup d'une anomalie à l'autre, selon la nature des connaissances requises.

Globalement, les approches mentionnées ci-haut proposent des techniques de détection automatique. Cependant, pour plusieurs types d'anomalies, ces approches présentent trois désavantages. Premièrement, les stratégies de détection sont toujours exprimées sous forme d'ensembles fixes ou de combinaisons de règles supposées détecter toutes les variations des anomalies de conception documentées. Deuxièmement, les règles basées sur les métriques utilisent souvent des valeurs seuil qui sont difficiles à définir. Finalement, la tâche de filtrer la liste des candidats détectés n'est pas discutés dans ces travaux.

2.3 Visualisation de logiciel

Même si l'objectif n'est pas spécifiquement la détection d'anomalies, d'autres approches utilisent la visualisation pour analyser des systèmes logiciels. Par exemple, Marcus *et al.* [24] utilisent la visualisation 3D pour assister l'utilisateur à l'inspection de code source C++. Les auteurs étendent la métaphore de *SeeSoft* [9] en lui ajoutant une troisième dimension. La figure 2.5 illustre les éléments de visualisation utilisés dans cette approche pour représenter le logiciel. Dans cette visualisation, les poly-cylindres représentent des lignes de code à l'intérieur d'un fichier de code source. Les attributs de chaque ligne sont associés à la couleur, la hauteur, la profondeur et la position du poly-cylindre.

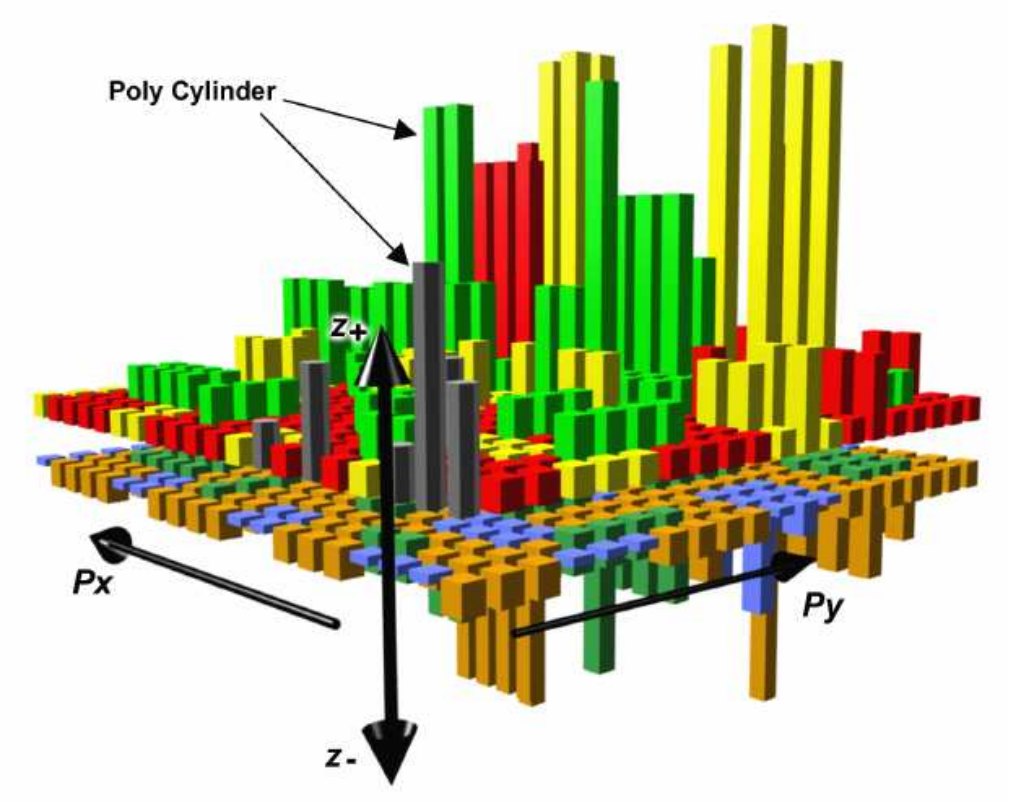


FIG. 2.5 – Éléments de visualisation utilisés par Marcus *et al.* [24].

Lanza et Ducasse [23] proposent une technique de visualisation de logiciels légère pour guider l'ingénieur logiciel durant les premières phases de rétro-ingénierie de logiciels de grande taille. La figure 2.6 montre la représentation d'un programme sous forme d'arbre en utilisant cette approche. Les attributs des noeuds de l'arbre sont le niveau de gris (qui varie de blanc à noir), la largeur, la hauteur et les coordonnées xy. Les arêtes représentent des relations d'héritage.

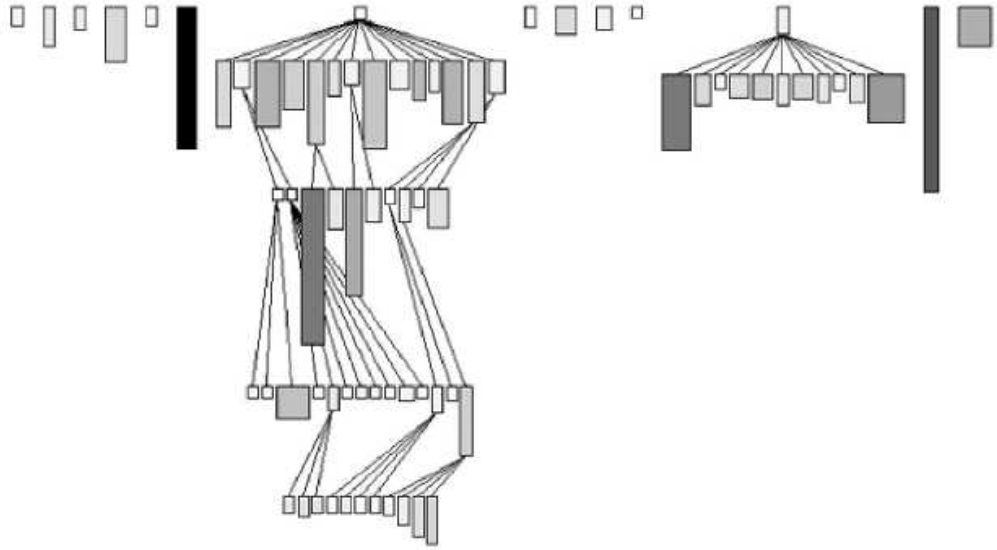


FIG. 2.6 – Vue poly-métrique d'un logiciel par Lanza et Ducasse [23].

Dans [8], Ducasse et Lanza étendent le travail amorcé dans [23] en ajoutant la visualisation de la structure interne des classes. De plus, les auteurs ont identifiés des patrons visuels représentant des situations récurrentes. Cette approche est représentée à la figure 2.7.

Plus récemment, Wettel et Lanza ont présenté un système de visualisation 3D qui utilise la métaphore de la ville pour la compréhension de logiciels [32]. Les auteurs proposent que la familiarité de la ville aide l'analyste à se repérer dans le logiciel.

Toutes ces techniques de visualisation présentent différentes vues globales de

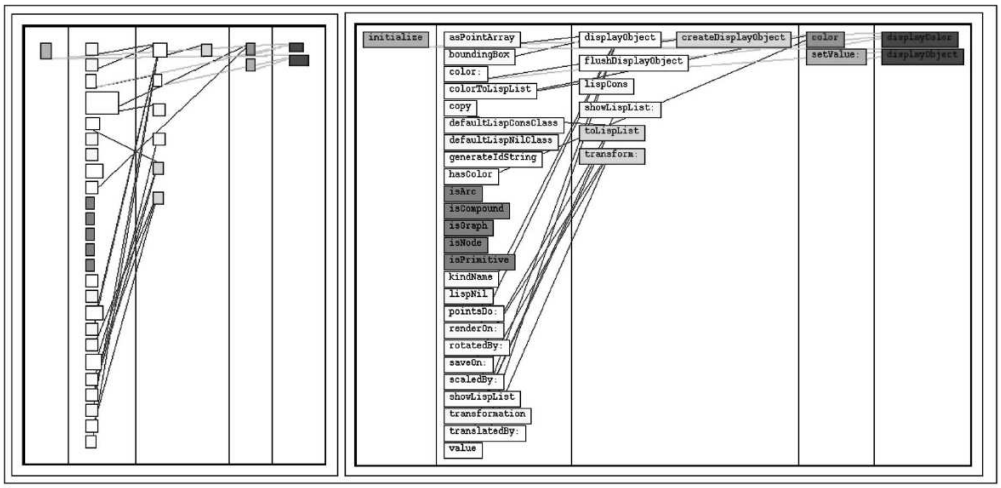


FIG. 2.7 – Visualisation *blueprint* d'une classe par Ducasse et Lanza [8].

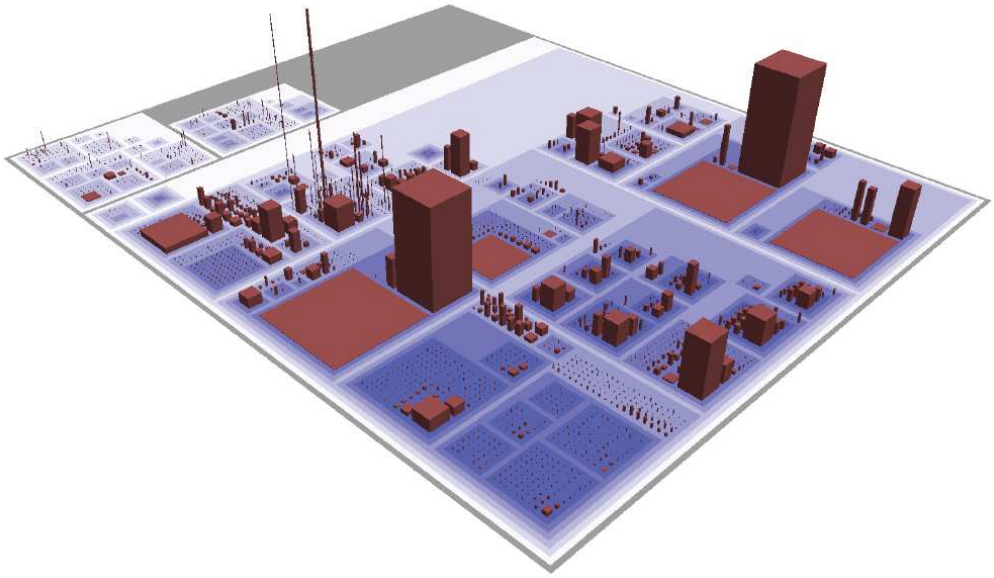


FIG. 2.8 – Le logiciel *ArgoUML* représenté avec la métaphore de la ville par Wettel et Lanza [32].

la conception d'un logiciel. Cependant, au meilleur de nos connaissances, elles n'offrent pas spécifiquement la possibilité de détecter des anomalies de conception comme les anti-patrons et les odeurs de code.

2.4 Conclusion

Dans ce chapitre, nous avons fait une revue des travaux reliés à l'approche présentée dans ce mémoire. Nous avons d'abord présenté les ouvrages qui définissent les anomalies de conception avec lesquelles nous travaillons. Ensuite, nous avons énuméré les techniques de détection se rapprochant le plus de la nôtre. Finalement, nous avons vu les plus récentes techniques de visualisation de logiciel. Dans le prochain chapitre, nous discutons des problématiques liées à la détection d'anomalies de conception.

CHAPITRE 3

PROBLÉMATIQUES DE DÉTECTION D'ANOMALIES

Bien qu'elles ne soient pas des défauts, *i.e.*, des violations de spécifications fonctionnelles, les anomalies de conception peuvent avoir un effet négatif sur plusieurs caractéristiques de qualité durant l'évolution du logiciel [3]. En dépit du progrès considérable accompli dans le champ de la qualité du logiciel, la détection et la correction d'anomalies demeurent aujourd'hui un moyen des plus concrets pour améliorer la qualité du logiciel. Cependant, la détection d'anomalies dans le code source est sujette à certaines difficultés inhérentes qui doivent être considérées. Dans ce chapitre, nous introduisons d'abord les définitions des anomalies de conception avec lesquelles nous illustrerons notre approche dans le reste de ce mémoire. Puis, en utilisant ces anomalies de conception, nous discutons de certains problèmes liés à la détection d'anomalies. Finalement, nous présentons les types d'informations du logiciel qui sont nécessaires à la détection d'anomalies.

3.1 Définition des anomalies de conception

Notre utilisation du terme "anomalie de conception" englobe les violations d'heuristiques de conception définies par Riel [31] et Martin [26], les anti-patrons décrits par Brown *et al.* [3], les odeurs de code documentées par Fowler *et al.* [12], ainsi que toute autre situation allant à l'encontre des heuristiques de qualité connues. Pour illustrer notre approche, nous introduisons quelques anomalies de conception : trois anti-patrons (*blob*, décomposition fonctionnelle, couteau suisse), deux odeurs de code (changement divergeant, *shotgun surgery*), ainsi qu'une violation d'heuristiques de qualité (classe mal placée).

3.1.1 *Blob*

Le *blob*, un anti-patron bien connu, se retrouve dans des architectures où une classe de grande taille monopolise le comportement, tandis que les autres classes encapsulent les données. Cet anti-patron est caractérisé par un diagramme de classes composé d'une classe contrôleur complexe et non-cohésive, associée à de simples classes de données [3].

En pratique, le *blob* peut prendre plusieurs formes. Comme exemple, nous pouvons citer le cas d'une classe qui devient de plus en plus complexe avec le temps. Cet exemple apparaît souvent dans des logiciels écrits à l'aide d'un environnement de développement qui oriente le développement autour d'interfaces graphiques. Ces programmes débutent souvent par un *formulaire* servant d'interface graphique, mais auquel on ajoute des fonctions au fur et à mesure que le besoin se présente. On se retrouve éventuellement avec une classe complexe et non-cohésive englobant à la fois l'interface graphique et le comportement du programme, se servant d'autres classes seulement pour stocker des données.

3.1.2 Décomposition fonctionnelle

L'anti-patron décomposition fonctionnelle apparaît lorsqu'une classe est conçue avec l'intention d'accomplir une fonction unique dans le système. La classe possède donc une grande méthode responsable de l'implémentation de cette fonction. La plupart des attributs de la classe sont principalement privés et utilisés à l'intérieur de la classe [3].

Cette anomalie prend la forme de code de style procédural (*i.e.*, une routine principale qui appelle des sous-routines pour exécuter le traitement) implémenté dans une architecture OO. Dans ce cas, chaque sous-routine est contenue dans un objet, créé uniquement pour cette raison. Nous avons donc une classe principale qui passe le contrôle à ces classes fonctionnelles pour exécuter le traitement. De telles classes portent souvent un nom qui dénote une fonction (*e.g.*, *CalculerInterets*).

3.1.3 Couteau suisse

L'anti-patron du couteau suisse se présente sous forme d'une classe ayant une interface excessivement complexe. Il s'agit d'une classe dont le concepteur n'a pas clairement défini le rôle. Le résultat est une classe ayant un grand nombre de méthodes publiques pour répondre à toutes sortes d'utilisations possibles. De telles classes sont difficiles à comprendre et à déboguer [3].

Des exemples réels de couteaux suisses peuvent inclure des centaines de signatures dans une même classe. On peut rencontrer cette anomalie sous forme d'une classe *utilitaire*. Une classe utilitaire renferme plusieurs opérations qui n'ont aucun lien entre elles. Suivant les principes de conception OO, ces opérations devraient plutôt être incluses dans les classes avec lesquelles elles agissent.

3.1.4 Changement divergeant

Le changement divergeant a lieu lorsqu'une classe est fréquemment changée de différentes façons et pour différentes raisons [12]. Il s'agit d'une classe qui doit souvent être modifiée lorsqu'un changement est apporté au système.

Cette anomalie peut se manifester de plusieurs façons dans un logiciel. Par exemple, une classe devant être modifiée chaque fois que l'on ajoute une nouvelle base de données au système, et qui doit aussi être changée lorsque l'on modifie une interface graphique serait un cas de changement divergeant. Il serait alors préférable de diviser cette classe en deux pour gérer ces deux préoccupations séparément.

3.1.5 *Shotgun surgery*

Le *shotgun surgery* est une odeur de code similaire à la précédente, mais de façon inverse. Cette anomalie a lieu lorsqu'un changement dans une classe nécessite plusieurs petits changements dans plusieurs classes différentes du programme. Lorsque les changements à faire sont répandus dans tout le programme, il est facile d'en oublier [12].

Le *shotgun surgery* peut apparaître dans des logiciels construits autour d'une classe jouant le rôle de noyau. La classe noyau contient la plupart des fonctions importantes du système, elle sera donc appelée à être changée fréquemment. Si ce noyau est utilisé par un grand nombre de classes dans le système, il est probable qu'un changement apporté au noyau nécessite plusieurs autres changements ailleurs.

3.1.6 Classe mal placée

Une classe mal placée va à l'encontre du principe de conception modulaire. Il s'agit d'une classe qui utilise et qui est utilisée par davantage de classes provenant d'autres packages que de son propre package. Si ces classes associées se trouvent principalement dans un même package, nous pourrions transférer la classe mal placée vers ce package [26].

Cette anomalie est le résultat d'une mauvaise architecture de packages. Les packages devraient englober des classes inter-reliées. Cependant en pratique, les packages ne sont pas toujours utilisés de cette façon. Les packages sont parfois utilisés pour délimiter le travail de différentes équipes de développement. Dans ce cas, il est probable que plusieurs classes se retrouvent mal placées.

3.2 Problèmes de détection

La détection d'anomalies de conception comporte certains problèmes inhérents la rendant difficile à appliquer en pratique. Dans cette section, nous discutons des problèmes liés à la taille de l'espace de recherche, la variabilité des occurrences, la définition des valeurs seuil et la dépendance au contexte.

3.2.1 Taille de l'espace de recherche

La taille de l'espace de recherche rend la détection d'anomalies difficile à implanter. En effet, plusieurs anomalies de conception peuvent être vues sous forme de

scénarios, où les rôles sont joués par des classes inter-reliées. Normalement, lorsque l'on détecte une anomalie comprenant m rôles dans un programme de n classes, on doit potentiellement vérifier $C_m^n = \binom{n}{m}$ combinaisons. Pour chacune de ces combinaisons, on doit vérifier si les classes satisfont les caractérisations de rôles et les patrons d'inter-relations.

3.2.2 Variabilité des occurrences

D'un autre point de vue, les anomalies de conception sont définies à plusieurs niveaux d'abstraction. Certaines anomalies sont définies en termes de structures de code, d'intentions du concepteur ou d'évolution de code. Bien que ces définitions soient dans plusieurs cas vagues et incomplètes, elles doivent être associées à des règles déterministes pour que la détection soit efficace. Et même quand c'est le cas, les occurrences d'une anomalie donnée sont rarement totalement conformes au patron tel que défini dans la littérature. Il existe un large éventail de variations. Un algorithme de détection doit être assez flexible pour couvrir cet éventail pour éviter les faux négatifs. L'intervention humaine durant le processus de détection peut pallier au problème de variabilité, mais seulement quand la taille du logiciel à analyser n'est pas trop grande.

3.2.3 Définition des valeurs seuil

La détection de nombreuses anomalies fait appel à des notions quantitatives, telles que la taille. Bien que l'on puisse mesurer la taille de la classe, la définition d'une valeur seuil appropriée n'est pas triviale. Une classe qui est considérée grande dans un programme donné pourrait être considérée moyenne dans d'autres. Cet aspect rend le processus de détection plus difficile à automatiser [25]. Pour être efficace, une méthode de détection doit tenir compte du contexte particulier d'un programme pour la prise de décision, au lieu d'utiliser des valeurs seuil fixes.

3.2.4 Dépendance au contexte

La détection d'anomalies n'est jamais totalement indépendante des programmes pour lesquels elle est pratiquée. Il est peu réaliste de penser qu'une méthode de détection d'anomalies puisse être utilisée sur n'importe quel programme, sans connaissances préalables de son contexte. Un logiciel peut être conçu d'une façon qui n'est pas conforme aux bonnes pratiques, mais qui devait être conçu de cette façon pour des raisons qui sortent du cadre de la conception à proprement dit (*e.g.*, à cause de considérations techniques ou financières). De telles raisons peuvent difficilement être intégrées dans un outil de détection automatique. Les méthodes de détection doivent être suffisamment flexibles pour considérer ces situations.

3.2.5 Conclusion

En considérant les points discutés dans cette section, nous concluons que pour un sous-ensemble d'anomalies de conception, la détection automatique présente plusieurs défis qui peuvent être amoindris en utilisant l'intervention d'un expert durant le processus de détection. Cependant, considérant la grande taille des logiciels industriels, il est difficile d'intervenir de manière efficace. Pour cette catégorie d'anomalies, nous proposons une approche et un outil qui peuvent être utilisés pour la maintenance de logiciel en tant que support pour l'inspection de code. Notre approche de détection visuelle est complémentaire aux approches de détection automatique. Bien qu'elle ne puisse être utilisée de manière systématique, *i.e.*, rechercher de façon exhaustive toutes les occurrences possibles, elle peut aider à détecter des anomalies qui sont difficiles à identifier automatiquement.

3.3 Représentation d'informations

La détection des anomalies introduites à la section 3.1 nécessite de l'information de quatre types : quantitatif, architectural, relationnel et sémantique. Ces types d'informations doivent être représentés pour qu'ils puissent être utilisés par

une approche de détection. Le reste de cette section définit ces quatre types d'informations. Dans le chapitre 4, nous décrivons comment nous représentons ces types d'informations avec VERSO dans le cadre de notre approche de détection.

3.3.1 Information quantitative

L'information quantitative d'un logiciel correspond aux différentes métriques qu'il est possible d'extraire du code source. Une des premières métriques logicielles est probablement le compte du nombre de lignes de code dans un programme (LOC). Dans le contexte des logiciels OO, on s'intéresse à compter le nombre d'opérations dans une classe, le nombre d'attributs d'une classe, le couplage entre les classes, etc. Les métriques OO les plus utilisées ont été proposées par Chidamber et Kemerer [4].

Plusieurs anomalies de conception sont définies/détectées en utilisant de l'information quantitative. *"Grande classe"* pour le *blob*, *"grande méthode"* pour la décomposition fonctionnelle et *"grand nombre de signatures d'interface"* pour le couteau suisse sont tous des exemples d'informations quantitatives nécessaire à la détection (voir la section 3.1).

3.3.2 Information architecturale

L'information architecturale fait référence à la façon dont sont organisées les classes dans un programme. Il n'est pas possible de mesurer de distance euclédienne entre deux parties de logiciel. Par contre, deux méthodes faisant partie d'une même classe peuvent être considérées plus "proches" que deux méthodes venant de classes différentes. De la même façon, deux classes sont plus rapprochées si elles se trouvent dans un même package, où si elles font partie du même arbre d'héritage.

Les définitions de la classe mal placée et de *shotgun surgery* comprennent deux éléments d'informations, respectivement *"classes venant de d'autres packages"* et *"répandu dans tout le programme"*. Ces éléments font référence à l'architecture de bas niveau, c'est-à-dire la structure du programme sous forme de modules/packages.

3.3.3 Information relationnelle

L'information relationnelle fait référence aux liens de dépendance entre les classes/modules d'un logiciel. Il peut s'agir de liens d'associations, d'invocations, de généralisation, etc.

Tel que mentionné dans la section 3.2, les anomalies de conception peuvent être vues comme des scénarios joués par des classes inter-reliées. Plusieurs types de relations sont utilisés dans ces scénarios. Par exemple, les définitions de *blob*, de la classe mal placée et de *shotgun surgery* incluent des relations d'associations et d'invocations.

3.3.4 Information sémantique

L'information sémantique fait référence à la connaissance du domaine qui n'est pas explicitement représentée dans le code source. Il pourrait s'agir, par exemple, d'un rôle donné à une classe par son concepteur. Bien que cette information n'existe pas textuellement dans le code source, connaître l'intention d'un concepteur aide à comprendre les choix de conception qui ont été pris durant le développement d'un logiciel.

Certaines anomalies sont définies et détectées en tenant compte de ce genre de connaissance. Pour l'exemple du *blob*, le fait qu'une classe joue le rôle de contrôleur combiné avec d'autres conditions peut améliorer l'efficacité de la détection de façon significative. De la même façon, dans la décomposition fonctionnelle, le fait qu'une classe implémente une fonction et non un concept est déterminant pour l'efficacité de la détection.

3.4 Conclusion

Dans ce chapitre, nous avons d'abord introduit les définitions des anomalies de conception qui seront utilisées dans le reste de ce mémoire. Puis, nous avons vu quelques problèmes liés à la détection d'anomalies. Finalement, nous avons présenté

les quatre catégories d'informations nécessaires à la détection d'anomalies. Dans le prochain chapitre, nous verrons de quelle façon ces catégories d'informations sont présentées à l'analyste dans VERSO.

CHAPITRE 4

REPRÉSENTATION D'INFORMATION DANS VERSO

Notre approche de détection d'anomalies est basée sur des fonctions fournies par notre système de visualisation de logiciels VERSO [21, 22]. VERSO génère des représentations 3D de logiciels de grande taille. Les entités du logiciel, représentées par des éléments graphiques 3D, sont distribués sur un plan 2D selon l'architecture de bas niveau du programme (*i.e.*, la structure du programme sous forme de modules/packages). Les représentations utilisent des données quantitatives (métriques) et structurelles (relations) obtenues en utilisant PADL [15], un outil de rétro-ingénierie et POM [16], un outil d'extraction de métriques. La Figure 4.1 illustre les étapes du procédé par lequel nous obtenons le fichier d'entrée pour VERSO à partir d'un programme donné.

Comme souligné au Chapitre 3, nos stratégies de détection utilisent de l'information classifiée en quatre catégories : quantitative, architecturale, relationnelle et sémantique. Le reste de ce chapitre présente comment ces catégories d'information sont présentées à l'analyste en utilisant VERSO et comment nous avons étendu VERSO pour permettre la détection d'anomalies de conception.

4.1 Information quantitative

L'information quantitative est capturée par les métriques extraites à partir du code. Pour une classe donnée, ses valeurs de métriques sont associées à des attributs graphiques qui seront visualisés par l'analyste. Les classes sont représentées par des boîtes 3D et les métriques sont associées avec trois attributs de la boîte 3D : la hauteur, la couleur et l'orientation (voir la Figure 4.2). Considérons par exemple une classe Java de 150 lignes de code (LOC = 150), ayant deux ancêtres (DIT = 2) et couplée avec 5 classes (CBO = 5). Une telle classe peut être représentée par

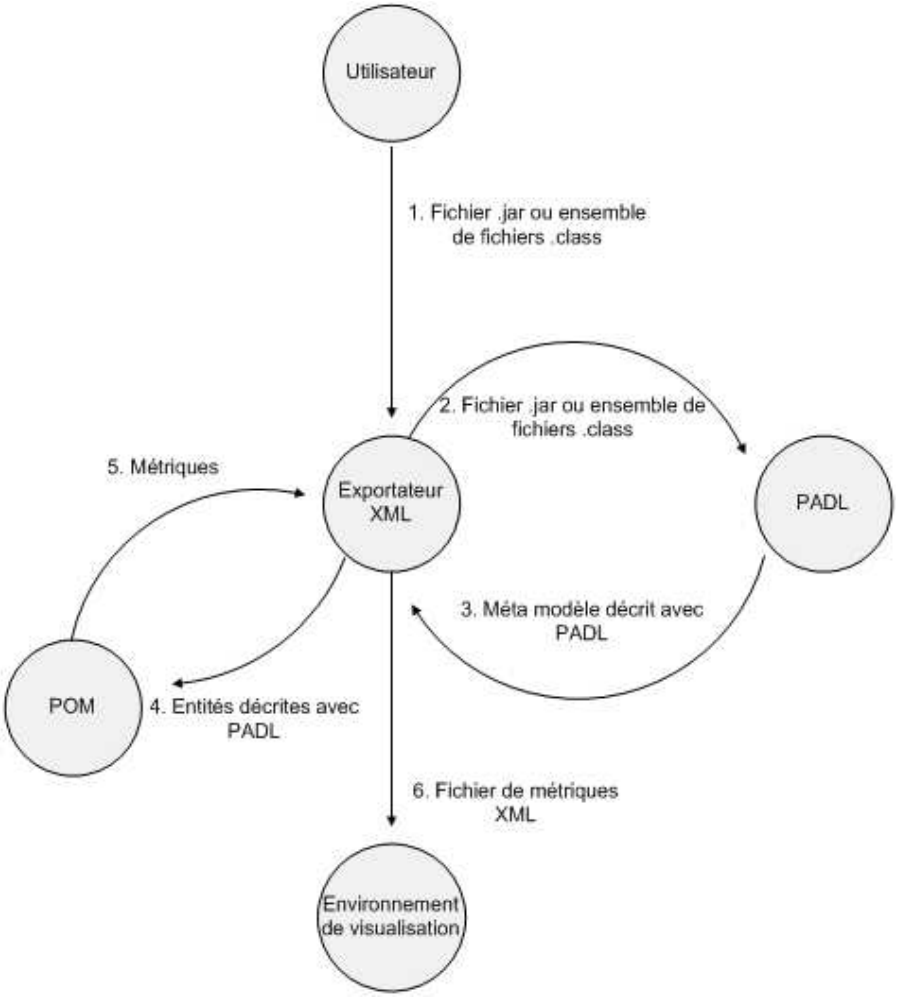


FIG. 4.1 – Pré-traitement pour générer le fichier d'entrée de VERSO.

une boîte 3D de hauteur h correspondant au LOC de 150, une couleur c (dans un spectre qui varie de bleu à rouge) correspondant au CBO de 5 et une orientation t (variant de 0 à 90 degrés, où 0 degré correspond à l'axe sud-nord) correspondant au DIT de 3. Ces attributs de la boîte 3D ont été choisis car ils sont faciles à distinguer, et présentent peu d'interférence entre eux, même avec plusieurs milliers de classes affichées simultanément [22]. Les interfaces sont représentées par des cylindres pour les distinguer des classes.

Les attributs graphiques des représentations de classes ont été choisis pour leur qualité au niveau de la perception du système visuel humain pour la visualisation d'une grande quantité d'information. Une justification de ces choix d'attributs a été présentée par Langelier [21] et Hassaine [17].

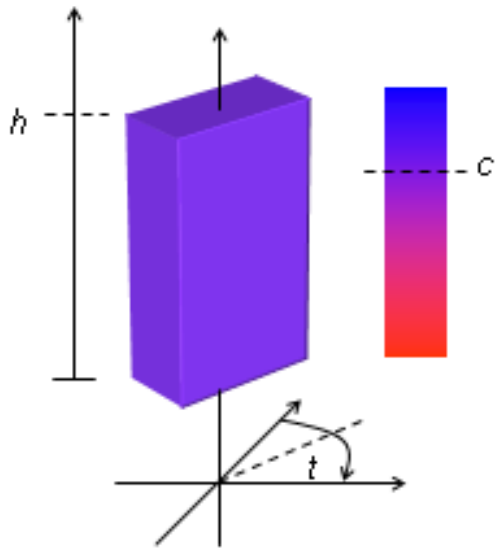


FIG. 4.2 – Représentation d'une classe.

Une fois que les métriques ont été calculées et associées aux attributs graphiques, un analyste peut évaluer visuellement si une condition de détection s'applique à une classe spécifique (*e.g.*, si la classe est grande). Cette évaluation peut être faite de la façon suivante. Pendant que l'analyste regarde toutes les boîtes (*i.e.*, les classes du programme analysé), il peut décider, en considérant le contexte global, si une classe a une grande valeur pour une métrique donnée comparativement aux autres.

À cette façon de procéder, nous avons ajouté diverses fonctions à VERSO pour améliorer la perception de l'information quantitative. Ces extensions sont le filtre de distribution, la modification dynamique du *mapping*, la composition de couleur ainsi que le *mapping* non-linéaire.

4.1.1 Filtre de distribution

Le filtre de distribution permet de changer l'apparence des classes selon leur position par rapport à la distribution d'une métrique donnée. Le filtre de distribution utilise la technique du *box plot* [10]. Ce filtre colore les classes selon leur position dans la distribution. Par exemple, les classes qui ont une valeur de métrique anormalement grande, *i.e.*, très éloignée du quartile supérieur, prennent la couleur rouge, alors que les classes ayant des valeurs situées entre les quartiles inférieur et supérieur sont colorées en vert. Les classes dans le quartile supérieur deviennent jaunes et celles dans le quartier inférieur deviennent bleues. Finalement, les classes ayant une valeur extrêmement petite apparaissent en bleu foncé. La Figure 4.3 montre l'application du filtre de distribution sur la métrique de couplage CBO pour les classes de la bibliothèque *Xerces*[1]. De plus, nous avons ajouté une fonction qui permet de visiter les classes à tour de rôle par ordre décroissant de la valeur de la métrique utilisée pour le filtre. Ainsi, lorsque l'on applique le filtre de distribution pour une métrique donnée, la classe ayant la plus grande valeur pour cette métrique est colorée en noir. Le nom de cette classe ainsi que sa valeur de métrique s'affichent alors automatiquement dans le haut de l'écran. Il suffit de choisir l'option *Suivant* dans le menu contextuel pour visiter la classe ayant la deuxième plus grande valeur et ainsi de suite. Cette option est utile dans le cas où l'on s'intéresse uniquement aux quelques classes ayant les plus grande valeurs pour une métrique donnée et d'éviter d'en oublier lors d'un traitement systématique.

[1] http ://xerces.apache.org/

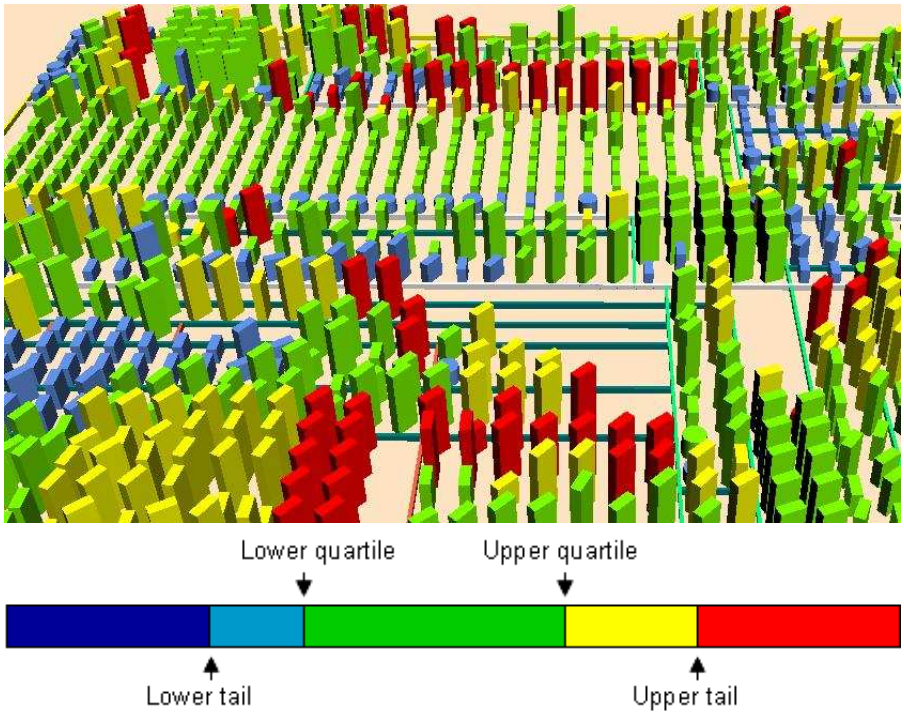


FIG. 4.3 – Une vue de *Xerces* avec le filtre de distribution appliqué pour la métrique CBO.

4.1.2 Marquage

Lors du développement de nos stratégies de détection d'anomalies, nous avons réalisé qu'il serait souvent utile de pouvoir marquer une classe, que ce soit pour indiquer qu'elle est une occurrence d'anomalie ou pour se rappeler qu'elle a suscité notre intérêt et que nous voulons l'inspecter davantage par la suite. Nous avons implémenté le marquage en colorant la face du dessus de la classe marquée. De cette façon, les classes marquées sont faciles à distinguer et l'information représentée par la couleur sur le reste de la classe est conservée.

Nous distinguons deux types de marquage : le marquage de sélection et le marquage d'anomalies. Le marquage de sélection colore le haut de la classe en bleu pâle. Ce type de marquage peut servir à se souvenir de candidats potentiels, ou de classes qui nécessitent davantage d'inspection. Le marquage d'anomalies sert à assigner aux classes des rôles dans des anomalies de conception. Lors du marquage,

l'utilisateur choisit le rôle à assigner parmi une liste lui étant présentée. Chaque classe peut être marquée de plus d'un rôle pour différentes anomalies. Le marquage d'anomalie colore le dessus de la classe en orange. Il est possible de sauvegarder les occurrences d'anomalies dans un fichier afin que l'analyste puisse poursuivre son travail dans une session de visualisation ultérieure.

4.1.3 Modification dynamique du *mapping*

Le *mapping*, dans le contexte de la visualisation scientifique, est le processus de transformation de données en représentations visuelles pour en améliorer la compréhension et l'analyse. Cette technique est utile pour extraire de l'information et identifier des tendances à partir de grandes quantités de données. Dans le cas de l'outil de visualisation VERSO, l'objectif est d'évaluer la qualité de systèmes logiciels en étudiant leur structure et les valeurs de métriques calculées pour leurs classes.

Lors de l'initialisation de VERSO, un *mapping* métrique–attribut graphique "par défaut" est suggéré à l'utilisateur : *Coupling Between Objects* (CBO)–couleur, *Weighted Methods per Class* (WMC)–hauteur et *Lack of Cohesion in Methods* (LCOM)–orientation. Il est possible de modifier ces associations métrique–attribut graphique avant de lancer la session de visualisation. Ces modifications sont utiles car elles permettent d'avoir des vues différentes du logiciel analysé. Cependant, une fois que la session de visualisation est lancée, il n'est plus possible de modifier le *mapping*. L'utilisateur doit à ce moment fermer la session de visualisation, modifier le *mapping* et relancer la session à nouveau. Ces étapes additionnelles causent une discontinuité qui a un impact négatif sur la perception de l'analyste.

Pour pallier à ce problème, nous avons ajouté à VERSO une fonction qui permet à l'utilisateur de changer le *mapping* sans interrompre la session de visualisation. L'utilisateur dispose maintenant d'une interface lui permettant de modifier le *mapping* et d'appliquer les modifications de façon dynamique à la session de visualisation courante. Ce mécanisme ajoute de la flexibilité au processus de vi-

sualisation d'un logiciel et favorise l'exploration. L'utilisateur peut aussi choisir un *mapping* parmi une liste de *contextes de détection*. Un contexte de détection est un *mapping* permettant de détecter une anomalie spécifique. Ce *mapping* correspond à la première étape des stratégies de détection présentées au Chapitre 5.

4.1.4 Composition de couleur

Comme nous avons vu au début de la Section 4.1, l'outil de visualisation VERSO utilise une boîte 3D pour représenter une classe d'un logiciel. Nous associons une métrique à chacun des trois attributs de cette boîte : sa hauteur, sa couleur et son orientation. La couleur varie de bleu à rouge, suivant la valeur de la métrique représentée. La Figure 4.4 illustre cette variation. La valeur minimale d'une métrique est représentée par la couleur bleu et la valeur maximale par la couleur rouge. Voici la fonction de *mapping* qui permet de déterminer la couleur de la classe :

$$rouge = 255 \times \left(\frac{M_v - M_{min}}{M_{max} - M_{min}} \right)$$

$$bleu = 255 - rouge$$

$$couleur = RVB\,(rouge, 0, bleu)$$

où M_v est la valeur de la métrique, M_{min} la valeur minimale que peut prendre la métrique, M_{max} la valeur maximale que peut prendre la métrique. Chacune des trois composantes de la couleur RVB peut prendre une valeur entre 0 et 255.

En utilisant la composition de couleurs [14, 20], nous avons ajouté à VERSO la possibilité d'associer trois métriques différentes à la couleur d'une classe. Chacune des métriques est représentée par l'une des trois composantes de l'espace couleur RJB : rouge, jaune et bleu. La composition de couleurs dans cet espace est soustractive, comme lorsque l'on mélange de la peinture. Les résultats de la composition devraient donc être plus intuitifs pour l'utilisateur. Pour chaque métrique représentée,

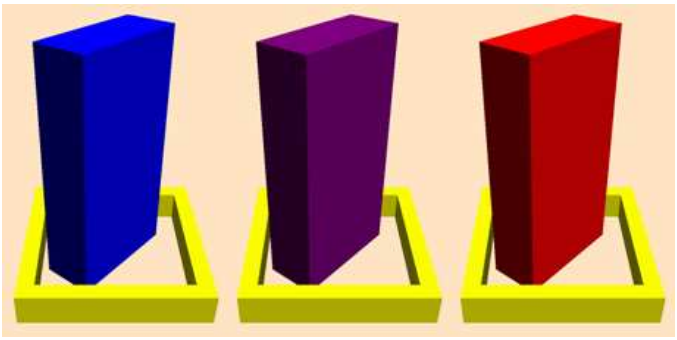


FIG. 4.4 – Couleur d'une classe variant graduellement de bleu à rouge.

l'utilisateur doit spécifier une valeur seuil. Si la valeur de la métrique est supérieure au seuil, la composante couleur (rouge, jaune ou bleu) qui lui est associée entrera dans la composition pour obtenir la couleur finale de la classe. Chaque composante sera donc soit présente ou absente de la composition. La classe peut ainsi prendre huit couleurs différentes : blanc, bleu, rouge, jaune, violet, orange, vert et marron. La Figure 4.5 illustre comment les couleurs se composent de façon additive dans l'espace RJB.

FIG. 4.5 – Illustration de la composition de couleurs en espace RJB.

VERSO peut être utilisé pour avoir une vue globale de programme. À ce niveau,

nous nous intéressons à la structure du système dans son entièreté, considérant l'apparence de l'ensemble des classes qui le composent. Nous pourrions, par exemple, vouloir déterminer si le logiciel a un trop fort couplage ou une complexité trop élevée. Nous mesurons le couplage à l'aide de la métrique CBO, la complexité par WMC et le manque de cohésion par LCOM. Si nous observons la classe encerclée en vert à la Figure 4.6, nous voyons qu'elle est fortement couplée, très complexe et peu cohésive.

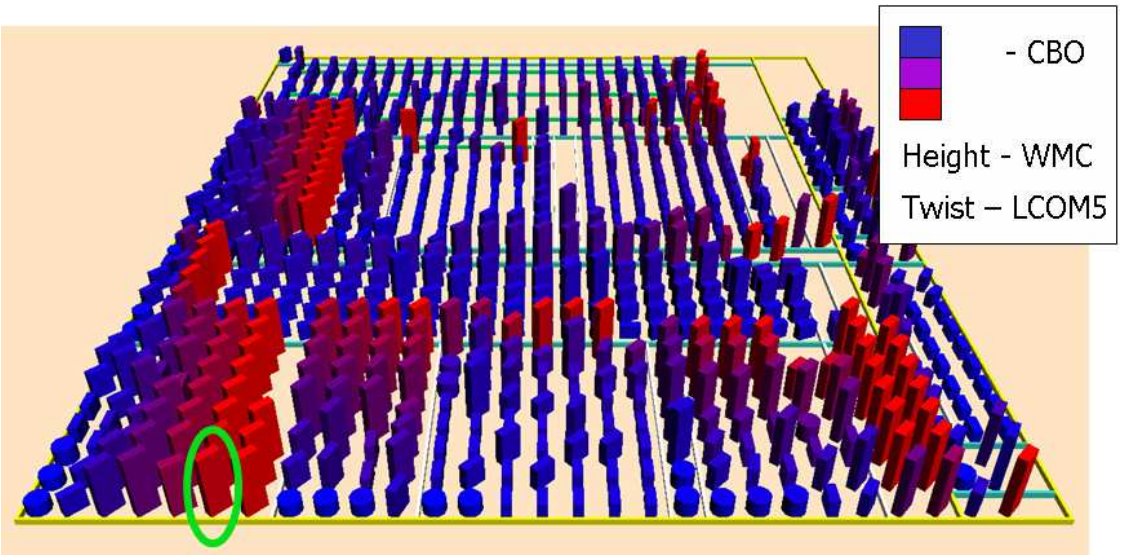


FIG. 4.6 – Vue globale d'un système avec le *mapping* bleu à rouge.

La Figure 4.7 représente la même scène, mais cette fois avec la composition de couleurs. Toutes les informations qui étaient représentées avec le *mapping* bleu à rouge sont toujours disponibles. De plus, deux informations additionnelles sont accessibles : la classe contient peu de déclarations de méthodes (*Number of Methods Declared* (NMD) faible) et elle est peu profonde dans l'arbre d'héritage (*Depth of Inheritance Tree* (DIT) faible).

Avec la composition de couleurs, nous pouvons augmenter le nombre de métriques représentées dans VERSO sans créer d'occultations au niveau de la perception de

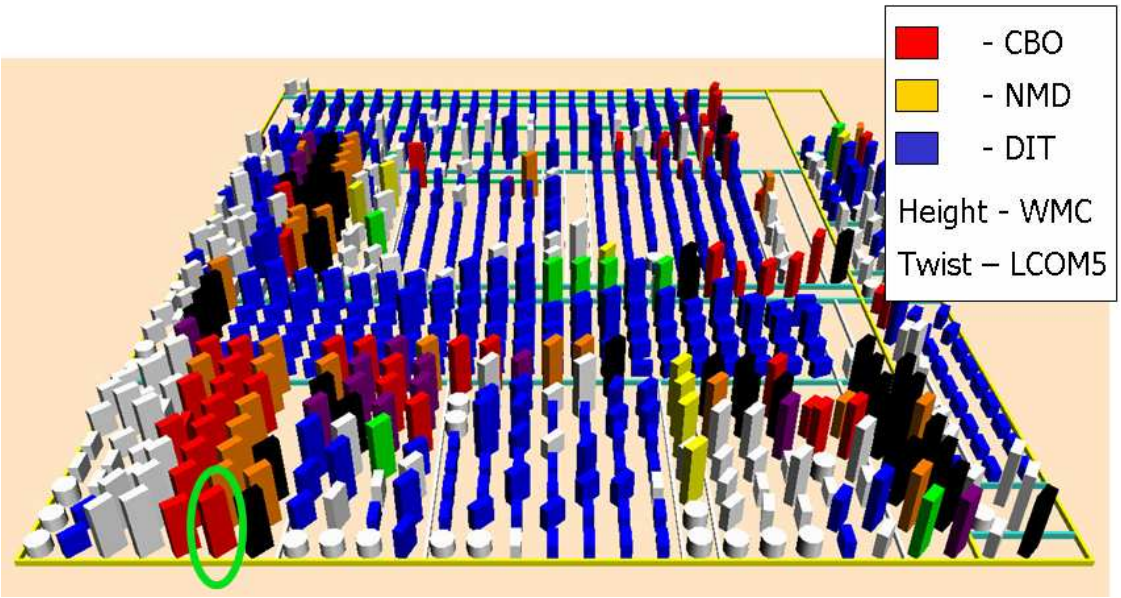


FIG. 4.7 – Vue globale d'un système avec *mapping* par composition de couleurs.

l'utilisateur. Comme trois métriques différentes peuvent être associées à la couleur avec cette technique, nous étendons donc à cinq le nombre de métriques représentées. Cette fonctionnalité rend possible la détection d'anomalies pour lesquelles un plus grand nombre de métriques est nécessaire.

4.1.5 *Mapping* non-linéaire

La performance d'un outil de visualisation comme VERSO dépend de sa capacité à représenter efficacement des données sous forme graphique. La fonction utilisée dans VERSO pour associer une métrique à un attribut graphique est linéaire. Il s'agit du type de *mapping* le plus intuitif. Chaque attribut graphique d'une représentation de classe a une intensité qui varie linéairement d'une valeur minimale à une valeur maximale. La couleur varie de bleu à rouge, l'orientation varie de 0 à 90 degrés et la hauteur varie d'une valeur plancher à une valeur plafond. Une représentation de classe apparaîtra complètement bleue si la métrique associée à la couleur a la plus petite valeur possible. Inversement, elle sera complètement rouge

si la métrique a la valeur maximale. Les classes pour lesquelles la valeur de métrique se trouve entre ces extrêmes seront représentées en utilisant l'interpolation linéaire. La Figure 4.8 montre le graphe de cette fonction.

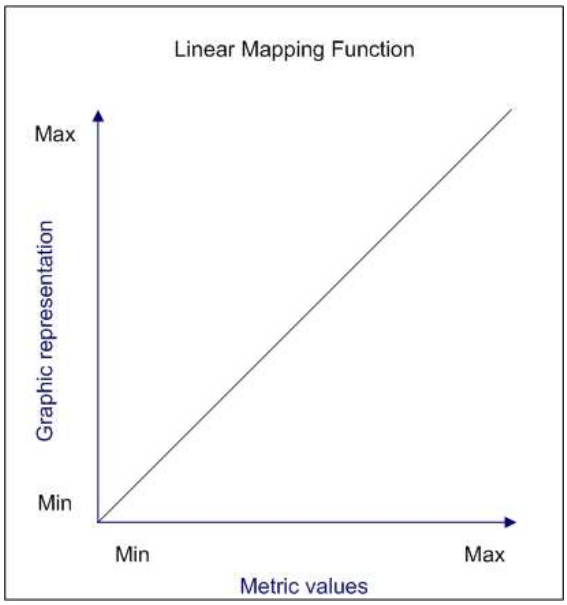


FIG. 4.8 – Fonction de *mapping* linéaire.

Cette méthode garantit que chaque classe est fidèlement représentée par rapport à sa position entre les deux *extrema* de la distribution d'une métrique donnée. Cependant, les ensembles de données liés à la structure des logiciels sont rarement uniformément distribués ; les échantillons de population sont souvent groupés dans des régions particulières de la distribution. Cela signifie qu'un ensemble de classes ayant des valeurs de métrique très rapprochées auront une représentation très similaire, au point où il peut devenir difficile de les distinguer. Même si ce *mapping* représente correctement les classes en fonction de leurs valeurs de métrique, il n'est pas utile si l'analyste ne peut pas facilement comparer les classes parce que leurs représentations sont trop similaires. Dans ces cas, une fonction de *mapping* qui utilise l'interpolation linéaire entre une valeur minimale et une valeur maximale n'est pas la technique la plus appropriée. Il est utile d'avoir davantage de contrôle

sur la fonction de *mapping* de façon à pouvoir donner plus ou moins de poids à certaines régions de la distribution de métriques. Dans le contexte de la détection d'anomalies, nous devons localiser des occurrences ayant des valeurs particulières (*e.g.*, des valeurs extrêmes). Nous voulons donc pouvoir modifier le *mapping* pour mettre l'emphase sur ces occurrences et les reconnaître plus facilement.

La Figure 4.8 montre que la fonction de *mapping* linéaire qui fait l'interpolation entre l'intensité minimale et l'intensité maximale, ce qui est un polynôme de premier degré. La fonction de *mapping* pourrait être plus flexible si elle était représentée par un polynôme de degré supérieur. Pour pallier à cette limitation, nous avons incorporé dans VERSO la possibilité de modéliser la fonction de *mapping* sous forme d'une courbe de *Bézier* [11], un polynôme de troisième degré. La courbe de *Bézier* est facile et intuitive à manipuler en déplaçant ses points de contrôle. Une courbe de troisième degré peut avoir jusqu'à deux points d'inflexion et est assez flexible pour représenter toutes sortes de fonctions de *mapping* non-linéaires. Nous avons fourni une interface que l'analyste peut utiliser pour modifier la forme de la courbe de *Bézier* en déplaçant ses points de contrôle, tel qu'illustré à la Figure 4.9.

Pendant que les points sont déplacés, la forme de la courbe est automatiquement mise à jour pour aider l'analyste à visualiser la fonction de *mapping* et l'apparence des représentations de classes est modifiée en conséquence. La formule pour évaluer la courbe de *Bézier* est définie comme suit :

$$Q(t) = (1-t)^3 P_1 + 3t(1-t)^2 P_2 + 3t^2(1-t) P_3 + t^3 P_4$$

où t varie de 0 à 1, et P_i est le i-ième point de contrôle.

Ramachandran [30] a décrit dans son travail sur l'esthétique l'effet *peak shift* comme l'accentuation d'une caractéristique rendant l'objet plus facilement reconnaissable que sa version originale. Cette théorie propose que si on est conditionné à répondre à un stimulus particulier, on réagira plus intensément si la caractéristique distinctive du stimulus est exagérée. La possibilité de modifier la fonction de *map-*

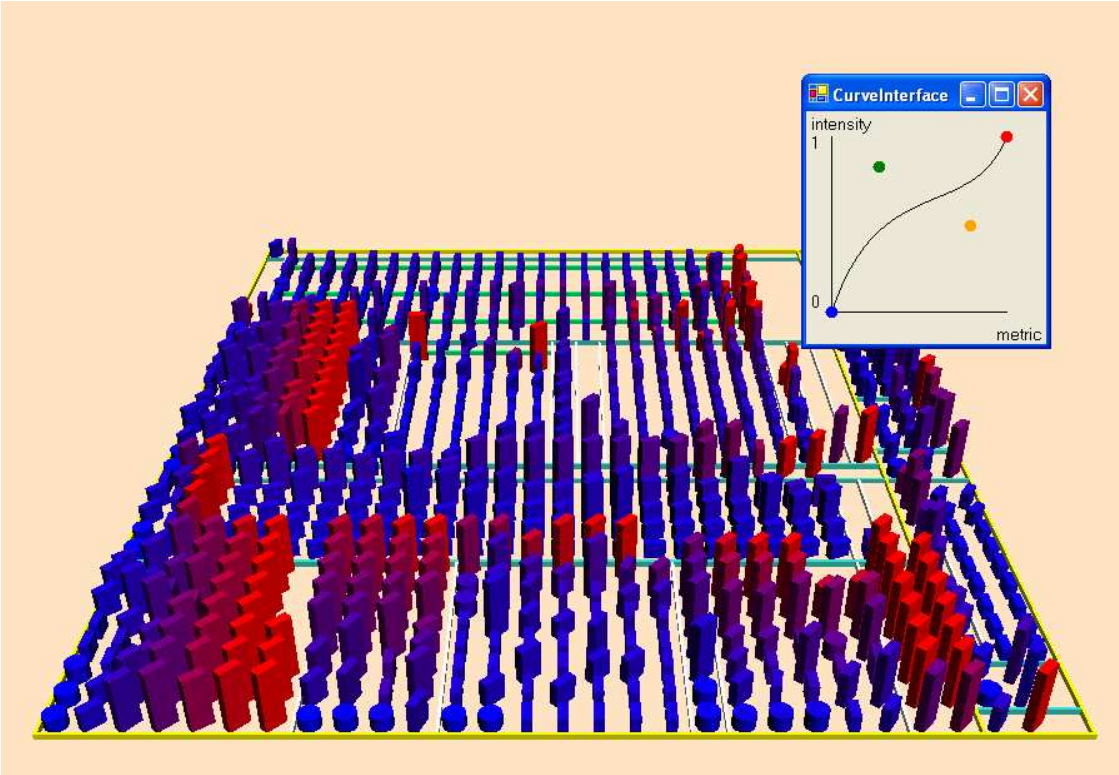


FIG. 4.9 – Interface pour manipuler la fonction de *mapping* par courbe cubique de *Bézier*.

ping dans VERSO exploite ce principe de *peak shift*. Dans notre cas, les caractéristiques distinctives d'une représentation de classe sont sa couleur, sa hauteur et son orientation. En ajustant la fonction de *mapping* et les valeurs minimales et maximales, l'analyste peut rendre plus apparents les attributs qu'il recherche.

4.2 Information d'architecture de bas niveau

Ce type d'information fait référence à l'architecture de bas niveau, c'est-à-dire la structure du programme sous forme de modules/packages. Comme nous sommes intéressés aux programmes *Java* ayant une structure de packages en arbre, nous organisons visuellement les classes en utilisant l'algorithme de placement *Treemap* [22]. La Figure 4.10 montre un exemple de visualisation de programme.

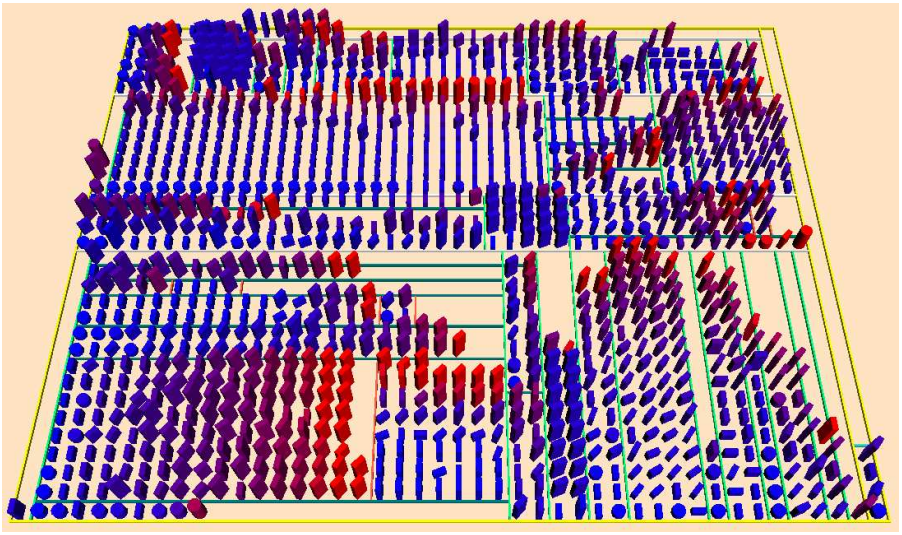


FIG. 4.10 – Représentation de *Xalan* (1194 classes).

L'algorithme de disposition considère le rectangle d'affichage (plan) comme la représentation du programme. Le rectangle est ensuite découpé en un certain nombre de tranches verticales, une pour chaque package principal. La taille de chaque tranche est calculée en fonction de la taille du package (*i.e.*, le nombre de classes contenues dans le package). À l'étape suivante, les nouveaux rectangles (tranches) sont découpés horizontalement pour diviser la surface d'affichage entre

les sous-packages. Le processus de découpage continue de la même façon, en alternant les séparations verticales et horizontales, jusqu'à ce que l'on atteigne les packages feuilles. Les boîtes 3D représentant les classes qui sont contenues dans un package donné sont placées sur le rectangle correspondant.

En observant la représentation d'un programme, il est facile de déterminer à quel package appartient une classe. Cette représentation peut informer l'analyste sur des problèmes de conception d'architecture, comme la prolifération de petits packages.

La prochaine sous-section présente des métriques de packages que nous avons implémentées, ainsi qu'une utilisation possible dans le contexte de la détection avec VERSO.

4.2.1 Métriques de package

Dans VERSO, nous représentons les logiciels au niveau de la classe. Les packages ne sont pas explicitement représentés. Cependant, en associant à chaque classe d'un package des métriques calculées pour le package en entier, nous verrons émerger des représentations de package, composées des classes qui en font partie.

Le *god package* est un exemple d'anomalie de conception détectable à ce niveau. Le *god package* englobe une grande quantité de classes, est peu cohésif (les classes composant le package travaillent peu ensemble) et est très utilisé par les autres packages du système. Pour détecter le *god package*, nous utilisons trois métriques : *Number Of Client Classes of a Package* (NOCCP), *Package Cohesion* (PC) et *Number of Client Packages* (NCP). Si on associe NOCCP à la hauteur de la classe, PC à son orientation et NCP à sa couleur variant de bleu à rouge, la stratégie de détection se résume à trouver des packages où les classes ont une taille élevée (NOCCP élevé), sont de couleur rouge (NCP élevé) et droite (PC faible). Nous pouvons observer un exemple d'un tel package en haut de la Figure 4.11.

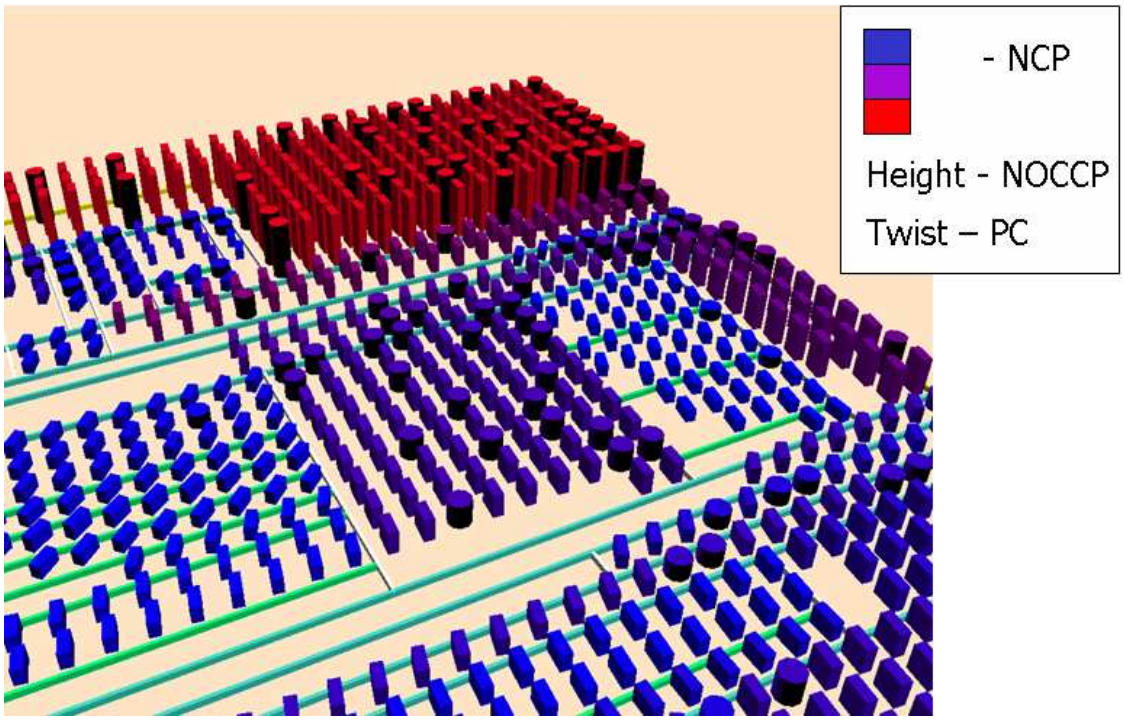


FIG. 4.11 – Occurrence de *god package*.

4.3 Information relationnelle

Les anomalies de conception peuvent être vues comme des scénarios joués par des classes inter-reliées. Plusieurs types de relations sont utilisés dans ces scénarios, en particulier les relations d'association et d'invocation.

Le reste de cette section est structuré comme suit. Nous présentons d'abord notre implémentation des filtres de visualisation pour VERSO. Ensuite, nous décrivons des expérimentations de filtrage en utilisant la suppression de la couleur, la transparence et la baisse de saturation.

4.3.1 Filtres relationnels

Dans notre outil de visualisation, nous utilisons des relations obtenues par rétro-ingénierie (association, agrégation, généralisation, implémentation, invocation, etc.). Étant donné que nous travaillons avec des programmes de grande taille (des milliers de classes inter-reliées), un affichage explicite et statique des relations pourrait facilement surcharger la perception de l'analyste, tel qu'illustré à la Figure 4.12. Pour contrer ce problème, nous avons développé un ensemble de filtres de structure, un pour chaque type de relation. Lorsqu'un filtre est appliqué sur une classe donnée, cette classe est colorée en vert, toutes les classes reliées conservent leur couleur originale, alors que toutes les autres classes non-reliées sont estompées, tel qu'illustré à la Figure 4.13.

Les filtres de structure peuvent aussi être combinés. Dans ce contexte, chaque filtre retourne un sous-ensemble de classes vérifiant un certain nombre de relations. Ce mécanisme de composition peut aussi être utilisé avec les filtres de distribution. Les filtres peuvent être combinés avec les opérateurs d'ensemble *union*, *intersection* et *différence*. L'interface usager de la composition de filtres est présentée à la Figure 4.14. Dans cet exemple, nous voyons la représentation du logiciel *Freenet*. Les classes qui apparaissent en rouge sont l'intersection des classes ayant une valeur extrêmement grande pour CBO et une valeur extrêmement grande pour WMC.

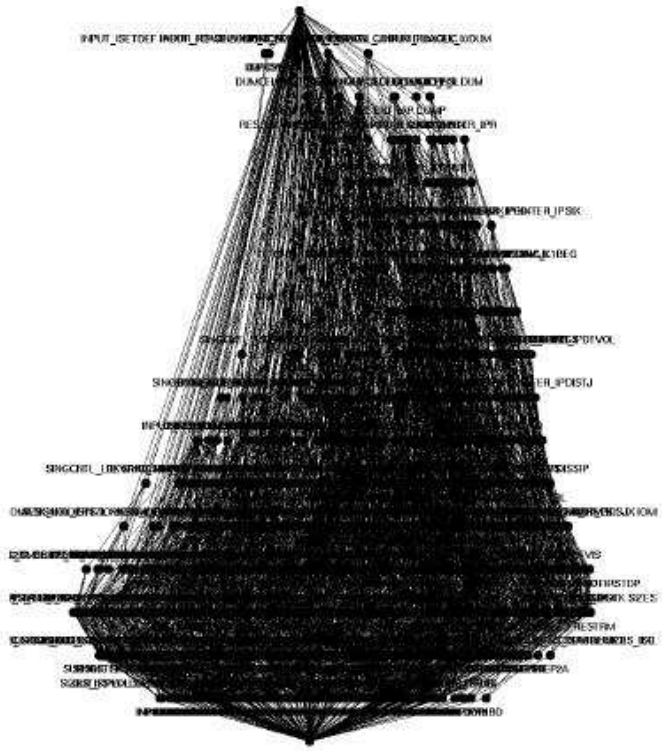


FIG. 4.12 – Diagramme de classes surchargé par l'affichage des relations.

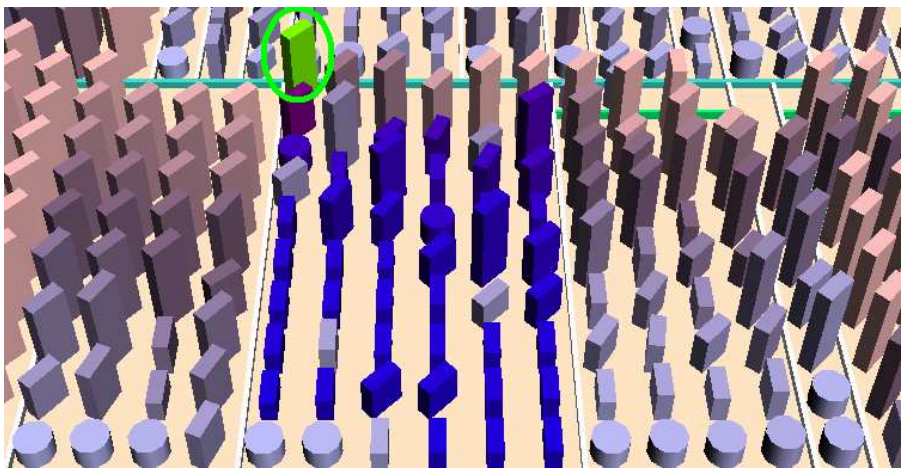


FIG. 4.13 – Un exemple de filtre de relation. Les classes ayant conservé leur couleur originale sont reliées à la classe verte par rapport à la relation sélectionnée.

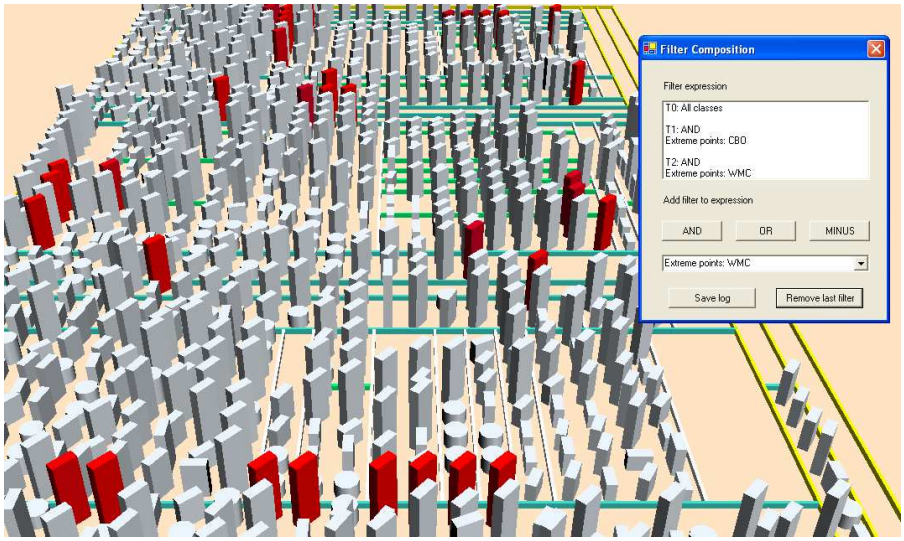


FIG. 4.14 – Composition de filtres de distribution.

4.3.2 Atténuation des représentations de classes

L'objectif des filtres de visualisation est de pouvoir mettre l'emphase sur un sous-ensemble de classes possédant certaines propriétés et d'estomper les autres. À cet effet, nous avons exploré trois possibilités : la suppression de la couleur, la transparence et la baisse de saturation.

La première option considérée lors de l'implémentation des filtres de visualisation fut de supprimer totalement la couleur des classes qui ne vérifient pas la relation pour laquelle on applique le filtre. C'est ce type d'atténuation qui est utilisé à la Figure 4.14. Nous voyons clairement que l'emphase est mise sur les classes ayant gardé leur couleur originale (dans ce cas-ci les classes rouges). Cependant, le désavantage de cette méthode est que l'information associée à la couleur des classes estompées est complètement perdue.

Par la suite, nous avons expérimenté l'atténuation des classes en les rendant transparentes. La Figure 4.15 montre le résultat de l'application d'un filtre de visualisation où la classes estompées ont perdu de l'opacité. La transparence a été réalisée par *alpha blending* : la couleur d'une classe est additionnée aux couleurs derrière. Nous n'avons pas retenu cette approche pour la version finale de VERSO.

En effet, cette approche devient moins efficace lorsqu'il y a un nombre important de classes les unes derrière les autres. Comme les couleurs sont additionnées dans l'espace RVB, nous obtenons rapidement la couleur blanc, ce qui n'était pas exactement l'effet de transparence recherché. Nous devons aussi donner une couleur foncée à l'arrière-plan. Autrement, toutes les classes apparaîtront très pâles, à cause du *blending* avec l'arrière-plan.

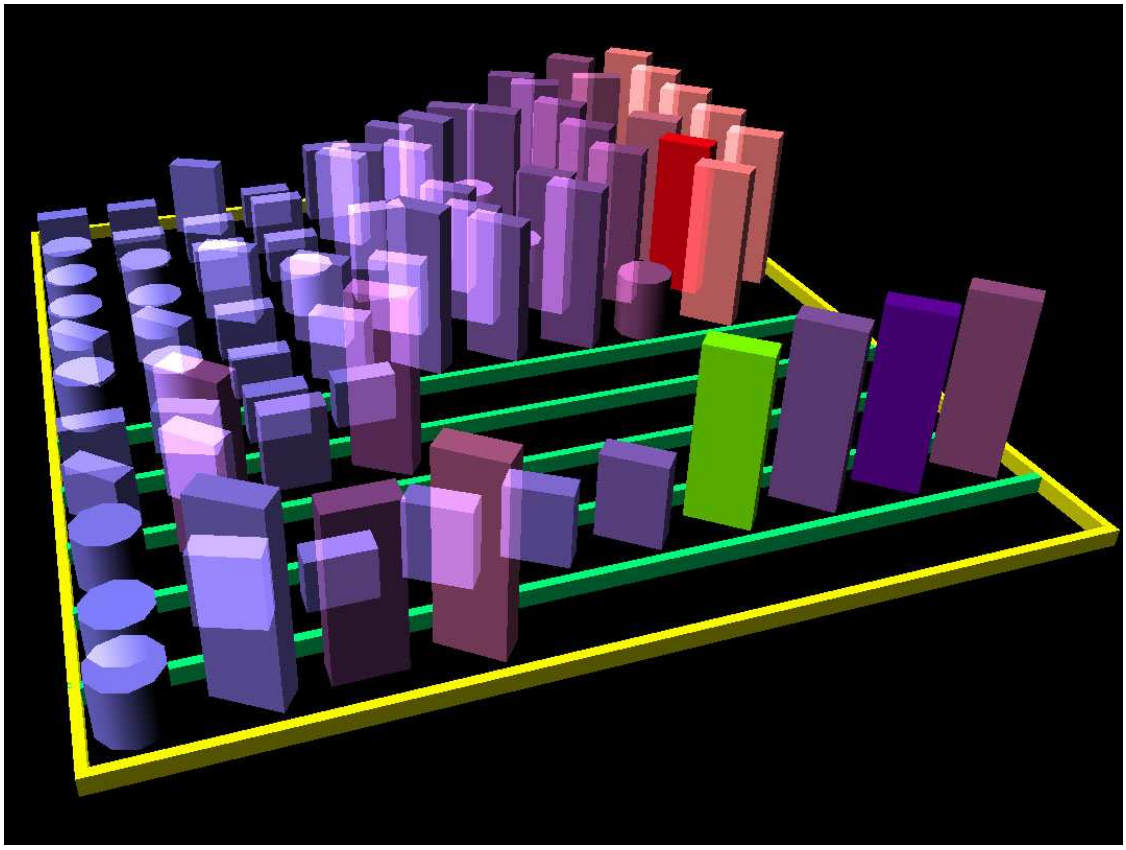


FIG. 4.15 – Application d'un filtre de visualisation en utilisant la transparence.

Finalement, nous avons choisi d'estomper les classes en réduisant la saturation de leur couleur. Cette option, présentée à la Figure 4.13, a été retenue dans la dernière version de VERSO. Sur la figure, la saturation est réduite de 30%. Cette technique permet à l'analyste de concentrer son attention plus facilement sur les

classes reliées, tout en conservant l'information associée à la couleur des classes estompées et donc une vue du contexte global du programme.

4.4 Information sémantique

L'information sémantique fait référence à la connaissance du domaine qui n'est pas explicitement représentée dans le code source. Dans le cas de l'anomalie décomposition fonctionnelle (voir la Section 3.1), par exemple, nous avons la notion de classes qui implémentent une fonctionnalité unique.

La prochaine sous-section présente un mécanisme que nous avons ajouté à VERSO permettant d'accéder au code source d'une classe.

4.4.1 Accès au code source

Lors de la détection avec notre outil de visualisation, l'analyste doit parfois déterminer si une classe joue un rôle donné. Il peut déduire cette information de deux façons. Premièrement, il peut regarder le nom de la classe. Dans plusieurs cas, le nom est suffisamment significatif pour indiquer le rôle de la classe. Si ce n'est pas suffisant, l'analyste peut parcourir le code source de la classe pour une analyse plus approfondie. L'analyste peut accéder au code source en sélectionnant la représentation de la classe avec la souris (voir la Figure 4.16). Bien que ce processus soit coûteux en temps, il est généralement fait après un premier filtrage, réduisant ainsi sa fréquence d'utilisation. En effet, lorsque nécessaire, l'information sémantique est souvent utilisée à la fin du processus de détection, pour accepter ou rejeter une classe candidate.

4.5 Conclusion

Dans ce chapitre, nous avons décrit de quelle façon les catégories d'information quantitative, architecturale, relationnelle et sémantique sont présentées à l'analyste dans VERSO. Nous avons aussi énuméré les extensions que nous avons apportées

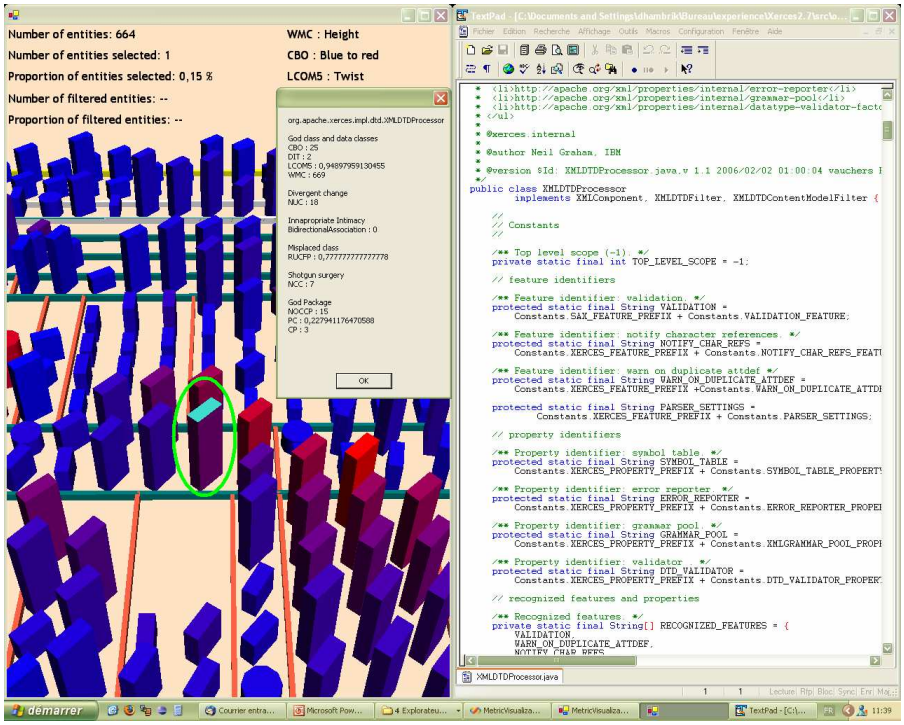


FIG. 4.16 – Accès au code source d'une classe à partir de VERSO.

à VERSO pour permettre la détection. Dans le prochain chapitre, nous voyons comment ces informations sont utilisées dans les stratégies de détection d'anomalies de conception.

CHAPITRE 5

DÉTECTION D'ANOMALIES

Ce chapitre décrit notre approche de détection semi-automatique d'anomalies basée sur la visualisation. Premièrement, nous classons les anomalies en trois types, selon leur définition en faisant le lien avec la détection. Ensuite, nous expliquons notre principe de détection général, à partir duquel nous dérivons des stratégies de détection pour des anomalies de conception spécifiques. Finalement, nous présentons des stratégies de détection pour les anomalies introduites dans la Section 3.1, avec des exemples concrets de détection dans des logiciels *open source*.

5.1 Types d'anomalies

Nous catégorisons les anomalies de conception en trois types. Le Type 1 inclut les anomalies pour lesquelles la définition est suffisamment précise pour être exprimée directement en termes de caractéristiques du code source. En d'autres mots, presque toute l'information nécessaire pour détecter ce type d'anomalies peut être trouvée dans le code. Le *blob* et la classe mal placée sont deux exemples d'anomalies du Type 1.

Le Type 2 fait référence aux anomalies pour lesquelles la définition inclut une intention de la part du programmeur. De telles intentions ne peuvent être trouvées directement par inspection du code source. Le couteau suisse est un exemple d'anomalie faisant partie de cette catégorie. Lorsque nous considérons si une classe est une occurrence de couteau suisse, nous devons juger si l'intention du programmeur a été de créer une classe avec un grand nombre de fonctionnalités diverses.

Les anomalies de Type 3 sont des problèmes de conception dont la présence peut seulement être confirmée en considérant l'évolution d'un programme. Le *shotgun surgery* est une anomalie de Type 3. La seule façon de s'assurer qu'un changement

fait à une partie du programme a nécessité plusieurs autres changements ailleurs est d'inspecter les versions du programme avant et après le changement. Cependant, pour les anomalies de Type 3, nous n'essayons pas de détecter leur présence, mais plutôt de détecter des symptômes dans le code source suggérant que l'anomalie pourrait se manifester dans le futur.

5.2 Principe de détection

Les anomalies de conception que nous étudions dans ce travail prennent la forme soit d'une classe unique ayant certaines propriétés non-désirées, soit une micro-architecture reliant un groupe de classes. Dans ce dernier cas, une des classes joue le rôle principal tandis que les autres jouent des rôles de support secondaires. Par exemple, pour l'anomalie du *blob* (Section 3.1) la classe contrôleur joue le rôle principal, et les classes de données jouent des rôles secondaires. Pour toutes les anomalies de conception impliquant plusieurs classes, nous avons trouvé qu'il était plus efficace de rechercher les classes principales en premier et de se concentrer sur les classes secondaires ensuite. L'explication est que les classes principales ont souvent des valeurs de métriques anormales, les rendant plus faciles à détecter avec notre outil de visualisation. De plus, le fait de voir les anomalies de micro-architecture comme des scénarios comprenant des rôles principaux et secondaires réduit considérablement l'espace de recherche pour l'analyste humain. Détecter des anomalies de conception comprenant m classes/rôles dans un programme de n classes peut mener à un grand nombre de combinaisons de classes à vérifier, dans le pire cas C_m^n. Cependant, lorsque l'on cherche des classes principales comme point d'entrée pour la recherche, le nombre de combinaisons de classes à vérifier est réduit à $n(m-1)$. En se basant sur cette observation, nous avons conçu un principe de détection général, à partir duquel nous dérivons des stratégies de détection pour des anomalies de conception spécifiques.

La première étape du principe de détection est de régler le *mapping*, *i.e.*, les

associations entre les valeurs de métriques et les attributs graphiques, en utilisant des métriques pertinentes pour caractériser les différents rôles de l'anomalie que l'on veut détecter. Ensuite, l'analyste localise toutes les classes candidates pour le rôle principal de l'anomalie. Ceci peut être fait en regardant la représentation du programme et en cherchant pour des classes ayant une certaine apparence (selon le *mapping* de métriques). Le filtre de distribution peut aussi être utilisé à cette deuxième étape pour visualiser la distribution de métriques et localiser les classes ayant des valeurs de métriques anormales. Toutes les classes candidates pour le rôle primaire sont marquées par l'outil de détection. Puis pour chaque classe marquée, l'analyste peut inspecter le nom et/ou le code source pour confirmer que la classe est réellement une occurrence du rôle principal. Si le rôle principal doit être supporté par des rôles secondaires (selon l'anomalie à détecter), une étape supplémentaire consiste à appliquer des filtres de relations. L'analyste considère l'apparence des classes reliées et décide s'il s'agit d'une occurrence de l'anomalie recherchée.

5.3 Exemples de détection d'anomalies

Le principe général défini plus haut peut être appliqué à un grand éventail d'anomalies. Dans le reste de cette section, nous présentons son application à six anomalies : le *blob*, la classe mal placée, la décomposition fonctionnelle, le couteau suisse, le changement divergeant et le *shotgun surgery*. Chaque anomalie est accompagnée d'un exemple concret. Les stratégies de détection utilisent les métriques présentées dans le Tableau 5.1. Les définitions détaillées des quatre premières peuvent être trouvées dans [2, 4, 19]. Nous définissons RCU à la sous-section 5.3.2.

5.3.1 *Blob* (Anomalie de Type 1)

5.3.1.1 Stratégie de détection

Étant une anomalie de Type 1, la définition du *blob* peut être entièrement exprimée sous forme de conditions de détection. Cette anomalie comporte deux types

Nom	Description
DIT	Niveau (profondeur) de la classe dans l'arbre d'héritage
LCOM5	Manque de cohésion dans les méthodes d'une classe
CBO	Nombre de classes interagissant avec une classe
NCC	Nombre de classes clientes
NPDM	Nombre de méthodes publiques déclarées d'une classe
NUC	Nombre de classes utilisées
WMC	Somme pondérée des complexités des méthodes d'une classe
RCU	Utilité relative d'une classe dans son package

Tab. 5.1 – Métriques de détection.

de rôles : la classe contrôleur qui implémente une grande partie du comportement et un ensemble de classes de données servant à ranger les données. La détection se fait en trois étapes : *mapping*, identification des classes contrôleur candidates et inspection des classes reliées (classes de données candidates). Trois métriques sont utilisées. Une valeur de WMC anormalement élevée indique une classe contrôleur potentielle, alors qu'un WMC faible indique des classes de données potentielles. LCOM5 est généralement élevé pour la classe contrôleur (implémentation de plusieurs fonctions indépendantes). Finalement, DIT est supposé être très faible pour les deux rôles (très peu d'ancêtres pour la classe contrôleur et les classes de données). La stratégie de détection pour le *blob* est résumée dans l'Algorithme 1.

5.3.1.2 Exemple

Un exemple d'occurrence de *blob* est présenté à la Figure 5.1. En haut, nous voyons le résultat du filtre de distribution pour la métrique WMC. La classe en noir a été sélectionnée comme classe contrôleur candidate. La classe est colorée en noir par le filtre de distribution car elle a la plus grande valeur courante de WMC (voir la Section 4.1.1 pour plus de détails sur le fonctionnement du filtre de distribution). Cette classe est candidate car elle a une valeur de WMC très élevée et également une valeur de LCOM5 très élevée (la classe est tournée). En bas, le filtre d'associations a été appliqué sur la classe candidate. La plupart des classes

Algorithm 1 Stratégie de détection du *blob*

1: Régler le *mapping* métriques–attributs graphiques à :
DIT-couleur, WMC-hauteur, LCOM5-orientation
2: Appliquer le filtre de distribution sur WMC
3: Localiser les classes candidates ayant une valeur de WMC extrêmement élevée
4: **for** chaque classe candidate *c* **do**
5: Inspecter les valeurs de LCOM5 et DIT pour *c*
(la boîte doit être bleue et relativement tournée)
6: Si nécessaire, inspecter le nom et le code de *c*
7: Appliquer le filtre d'associations "sortantes" sur *c*
8: Inspecter les classes reliées à *c*
(elles doivent être petites (WMC faible), droites (très cohésives), et bleues (DIT faible))
9: **if** toutes les étapes précédentes sont positives **then**
10: Sauvegarder l'occurrence de *blob*
11: **end if**
12: **end for**

associées (celles qui sont toujours colorées) sont bleues, petites et droites, ce qui signifie qu'il s'agit de classes de données. Le *blob*, *org.eclipse.swt.internal.win32.OS*, a été détecté dans *RSSOwl*, un lecteur de RSS *open source*. Cette classe joue le rôle d'interface de communication entre la librairie d'interface graphique *Standard Widget Toolkit* (SWT) de *Java* et la librairie d'interface graphique du système d'exploitation *Windows*. Il est admis que ce rôle soit joué par une classe unique. Cependant, dans ce cas-ci, la classe *OS* fait tout le traitement par elle-même et interagit seulement avec des classes de données. Il s'agit donc bien d'une occurrence de *blob*.

5.3.2 Classe mal placée (Anomalie de Type 1)

5.3.2.1 Stratégie de détection

Une classe mal placée est une classe qui interagit majoritairement avec un ensemble de classes situées dans un package autre que le sien [26]. Pour détecter cette anomalie, nous définissons la nouvelle métrique RCU qui donne l'utilité relative

d'une classe dans son package. RCU est la proportion de classes qui utilisent ou qui sont utilisées par la classe considérée, se trouvant dans des packages autres que le sien. Un RCU= 0 signifie que la classe interagit seulement avec des classes de son propre package, alors que RCU= 1 indique que la classe interagit uniquement avec des classes d'autres packages.

La détection s'effectue en trois étapes : (1) régler le *mapping*, (2) identifier les classes ayant une valeur de RCU proche de 1, une grande complexité et une faible cohésion et (3) inspecter l'emplacement des classes reliées pour vérifier si elles se trouvent majoritairement dans un même package. La stratégie de détection pour la classe mal placée est résumée dans l'Algorithme 2.

Algorithm 2 Stratégie de détection de classe mal placée

1: Régler le *mapping* métriques–attributs graphiques à :
RCU-couleur, WMC-hauteur, LCOM5-orientation
2: Localiser les boîtes rouges (RCU≈ 1), grandes (complexes), et tournées (non cohésives)
3: **for** chaque classe candidate c **do**
4: Appliquer le filtre d'associations "entrant/sortant" sur c
5: Inspecter les classes associées avec c
(elles doivent être pour la plupart localisées dans un même package)
6: **if** toutes les étapes précédentes sont positives **then**
7: Sauvegarder l'occurrence de classe mal placée
8: **end if**
9: **end for**

5.3.2.2 Exemple

Un exemple de classe mal placée est présenté dans la Figure 5.2. Dans la partie du haut, la classe encerclée en vert a été sélectionnée comme candidate car elle est rouge vif (RCU près de 1), grande (grande valeur de WMC) et tournée (grande valeur de LCOM5). En bas, on peut voir le résultat du filtre d'associations "entrantes/sortantes". La majorité des classes reliées (celles dont la couleur est restée saturée) sont concentrées dans un autre package, à droite de l'image.

Cette classe candidate a été trouvée dans le logiciel *Art of Illusion*. Cette classe est mal placée car elle n'a aucune interaction avec les classes de son package. Le fait de déplacer cette classe dans le package où sont contenues la majorité des classes reliées réduirait les dépendances inter-packages dans le programme.

5.3.3 Décomposition fonctionnelle (Anomalie de Type 2)

5.3.3.1 Stratégie de détection

La décomposition fonctionnelle peut être détectée en observant les conséquences des intentions du concepteur dans le code. Dans ce cas-ci, le concepteur crée des classes qui implémentent une fonctionnalité unique [3]. Les conséquences d'une telle conception sont que les classes n'utilisent pas le mécanisme d'héritage (DIT très faible) et n'ont pratiquement qu'une seule méthode publique déclarée (NPDM$\approx$ 1). Cette méthode est très grande et complexe, ce qui influence la complexité de la classe (WMC élevé). La détection de ces symptômes est résumée dans l'Algorithme 3.

Algorithm 3 Stratégie de détection de la décomposition fonctionnelle

```
1: Régler le mapping métriques–attributs graphiques à :
   NPDM-Couleur, WMC-hauteur, DIT-orientation
2: Localiser les classes qui sont grandes (complexes), bleues (NPDM très faible),
   et presque droites (DIT très faible)
3: for chaque classe candidate c do
4:     Inspecter le nom et le code source de c
       (nom de classe indiquant une fonction, plusieurs attributs privés, etc.)
5:     if toutes les étapes précédentes sont positives then
6:         Sauvegarder l'occurrence de décomposition fonctionnelle
7:     end if
8: end for
```

5.3.3.2 Exemple

Des exemples de décomposition fonctionnelle détectés dans *ArgoUML* sont montrés à la Figure 5.3. Les classes encerclées en vert ont été identifiées comme classes

candidates. Elles sont droites (faible valeur de DIT), bleues (faible valeur de NPDM) et grandes (valeur de WMC élevée). L'inspection du code source a révélé que ces classes implémentent les fonctions d'initialisation (*checklist.Init* et *critics.Init*), ainsi qu'une fonction de génération (*critics.ChildGenUml*).

5.3.4 Couteau suisse (Anomalie de Type 2)

5.3.4.1 Stratégie de détection

La détection du couteau suisse est similaire à celle de l'anomalie précédente. Les conséquences possibles des intentions du concepteur dans le code sont des classes qui manquent de cohésion (LCOM5 élevé), ayant un grand nombre de méthodes publiques déclarées (NPDM élevé), utilisées par plusieurs classes (NCC élevé - nombre de classes clientes). La détection d'occurrences de couteau suisse est résumé dans l'Algorithme 4.

Algorithm 4 Stratégie de détection du couteau suisse.

```
1: Régler le mapping métriques–attributs graphiques à :
   NPDM-couleur, NCC-hauteur, LCOM5-orientation
2: Appliquer le filtre de distribution sur NPDM
3: Localiser les classes qui sont grandes (NCC élevé), rouges (NPDM très élevé),
   et tournées (LCOM5 élevé)
4: for chaque classe candidate c do
5:    Inspecter le code source de c
      (nom de méthodes indiquant des opérations non-reliées)
6:    if toutes les étapes précédentes sont positives then
7:       Sauvegarder l'occurrence de couteau suisse
8:    end if
9: end for
```

5.3.4.2 Exemple

Des exemples de couteau suisse trouvés dans *ArgoUML* sont montrés dans la Figure 5.4. Toutes les classes encerclées en vert ont été détectées comme candidates de couteau suisse car elles sont grandes (valeur de NCC élevée), rouges (valeur

de NPDM élevée) et tournées (valeur de LCOM5 élevée). L'inspection du code a confirmé que ces classes candidates sont des occurrences de couteau suisse. Les classes sont *Critic*, *Designer*, *ListSet* et *ProjectBrowser*. Ces classes sont des occurrences de couteau suisse car elles ne représentent pas des concepts ou abstractions dans le programme et contiennent chacune un grand nombre de méthodes publiques qui n'ont pas de liens entre elles.

5.3.5 Changement divergeant (Anomalie de Type 3)

5.3.5.1 Stratégie de détection

Étant une anomalie de Type 3, le changement divergeant ne peut être détecté en analysant une version unique d'un programme. Une détection précise requiert l'analyse de plusieurs versions. Cependant, nous pouvons détecter des situations qui peuvent causer le changement divergeant. Une situation typique est une classe complexe (WMC élevé) et non-cohésive (LCOM5) qui utilise un grand nombre de classes (NUC élevé) répandues partout dans le programme. Chaque fois qu'une de ces classes reliées est changée, il existe une probabilité que la classe complexe qui l'utilise doive aussi être changée. La stratégie pour détecter cette situation est définie dans l'Algorithme 5.

Algorithm 5 Stratégie de détection de changement divergeant.

1: Régler le *mapping* métriques–attributs graphiques à :
NUC-couleur, WMC-hauteur, LCOM5-orientation
2: Appliquer le filtre de distribution sur NUC
3: Localiser les classes rouges (NUC très élevé), grandes (WMC élevé) et tournées (LCOM5 élevé)
4: **for** chaque classe candidate c **do**
5: Appliquer le filtre d'associations "sortantes" sur c
6: Inspecter les classes associées à c
(elles doivent être répandues dans tout le programme)
7: **if** toutes les étapes précédentes sont positives **then**
8: Sauvegarder l'occurrence de changement divergeant
9: **end if**
10: **end for**

5.3.5.2 Exemple

La Figure 5.5 présente un exemple de situation détectée dans *ArgoUML* pouvant mener à un changement divergeant. En haut, nous voyons le résultat du filtre de distribution appliqué sur NUC. La classe encerclée en noir (*argouml.uml.ui.TabProps*) a été identifiée candidate de changement divergeant car elle a une valeur de NUC très élevée comparativement au reste du programme (101 classes, ce qui représente environ 7% du nombre total de classes dans le programme). Le résultat du filtre d'associations sortantes est montré au bas de la figure. Il montre que les classes reliées sont répandues dans plusieurs parties différentes du programme. Cette occurrence de changement divergeant représente l'onglet contenant le panneau de propriétés du logiciel *ArgoUML*. Cette classe "écoute" un grand nombre de composants dans le système pour pouvoir mettre à jour le panneau de propriétés dès qu'il y a un changement. Un ajout de composant dans le système est donc susceptible d'impliquer un changement dans la classe *org.argouml.uml.ui.TabProps*.

5.3.6 *Shotgun surgery* (Anomalie de Type 3)

5.3.6.1 Stratégie de détection

À l'inverse de l'anomalie précédente, une classe qui est utilisée par un grand nombre de classes répandues partout dans le programme est une situation pouvant mener à l'anomalie *shotgun surgery*. Si cette classe est modifiée, les classes qui l'utilisent doivent potentiellement être modifiées également. Elle peut être détectée en utilisant l'Algorithme 6.

5.3.6.2 Exemple

La Figure 5.6 montre une situation pouvant mener au *shotgun surgery* détectée dans le logiciel *Freenet*. En haut de la figure, la classe encerclée en vert en bas à droite a été identifiée comme un rôle principal à cause de sa valeur de NCC extrêmement élevée. En bas, le filtre d'associations "entrantes" montre que la classe

Algorithm 6 Stratégie de détection de *shotgun surgery*.

1: Régler le *mapping* métriques–attributs graphiques à :
NCC-couleur
2: Localiser les classes candidates complètement rouges (valeur de NCC extrêmement élevée)
3: **for** chaque classe candidate c **do**
4: Appliquer le filtre d'associations "entrantes"
5: Inspecter les classes associées à c
(elles doivent être répandues dans tout le programme)
6: **if** toutes les étapes précédentes sont positives **then**
7: Sauvegarder l'occurrence de *shotgun surgery*
8: **end if**
9: **end for**

candidate au rôle principal *freenet.Core* est utilisée par environ 30% du nombre total de classes dans le programme et que ces classes sont répandues partout dans le programme. *Freenet* est un logiciel qui permet de publier et obtenir de l'information sur Internet de façon anonyme. Toutes les communications de ce réseau décentralisé se font au moyen de passage de noeuds encryptés. La classe *freenet.Core* sert de classe enveloppante pour ces noeuds par rapport au reste du système. Un changement apporté à *freenet.Core* peut donc nécessiter un grand nombre de changements dans tout le système.

5.4 Conclusion

Dans ce chapitre, nous avons décrit les trois types d'anomalies que nous détectons avec notre approche. Ensuite nous avons présenté notre principe général de détection qui contribue à diminuer l'espace de recherche pour l'analyste. Finalement, nous avons présenté les stratégies de détection pour six anomalies de conception, avec des exemples dans des systèmes réels. Dans le prochain chapitre, nous présentons les résultats de deux études de cas où des sujets devaient mettre en oeuvre ces stratégies de détection.

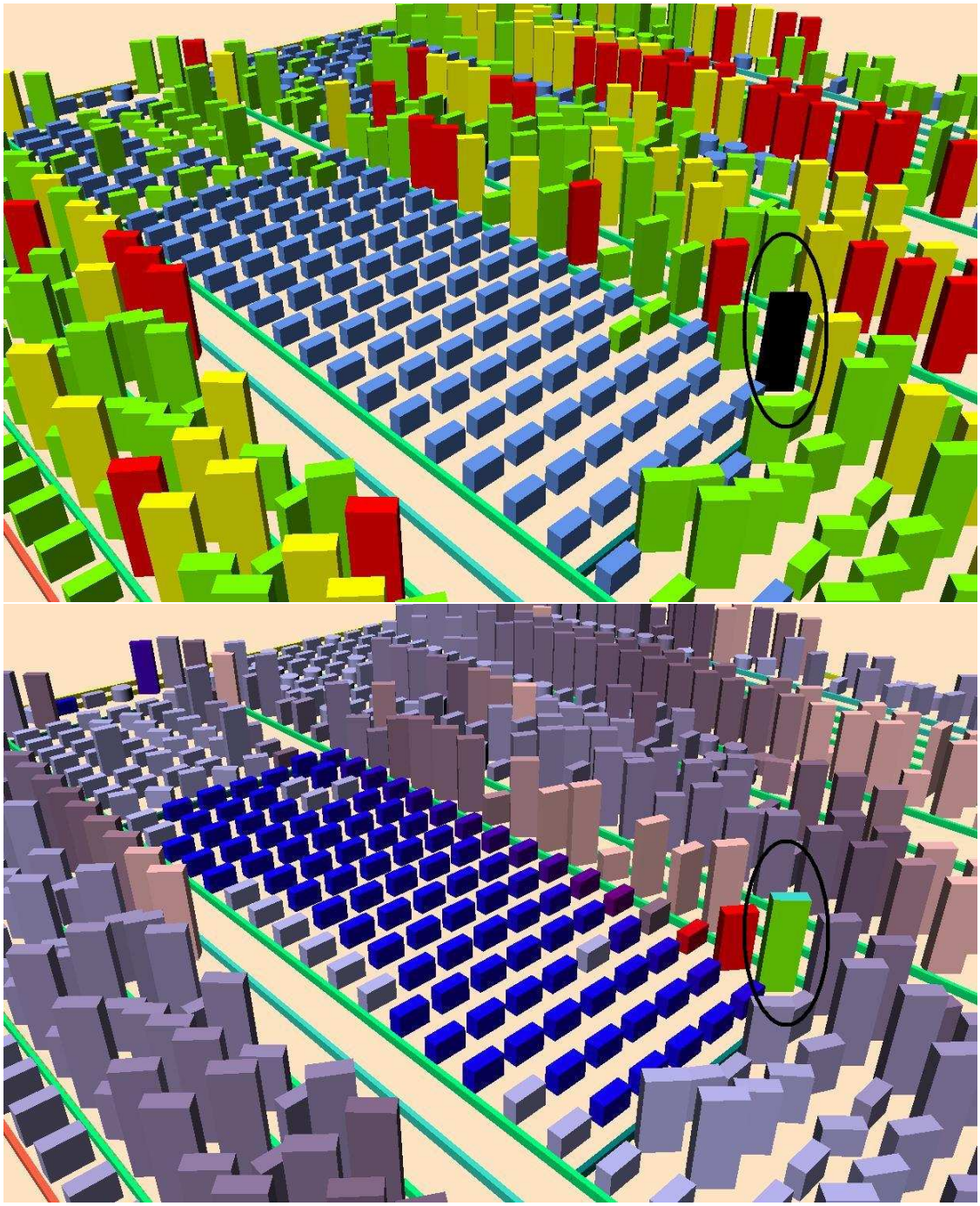


FIG. 5.1 – Un exemple de *blob* découvert dans *RSSOwl*. (Haut) Filtre statistique appliqué sur WMC. (Bas) Filtre d'associations appliqué sur la classe encerclée.

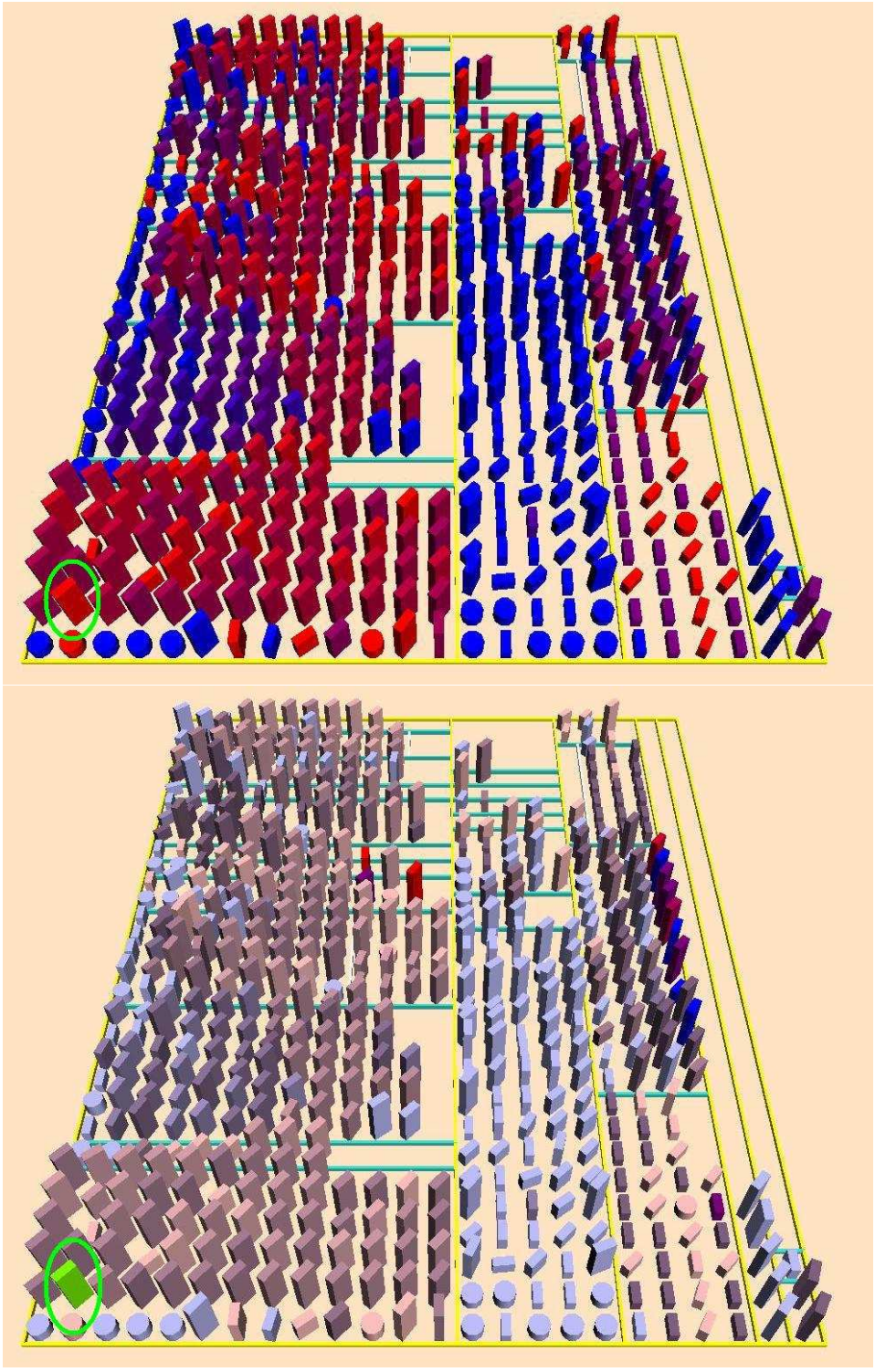


FIG. 5.2 – Un exemple de classe mal placée découverte dans *Art of Illusion*. (Haut) *Mapping* initial pour détecter la classe de rôle principal. (Bas) Filtre d'associations appliqué sur la classe encerclée.

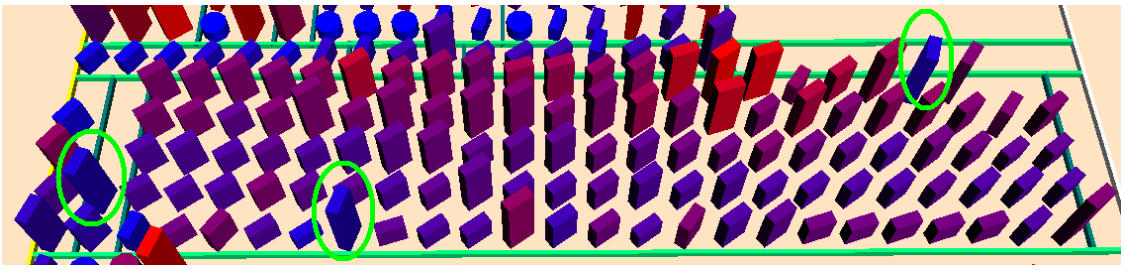


FIG. 5.3 – Exemples de décomposition fonctionnelle détectés dans *ArgoUML*.

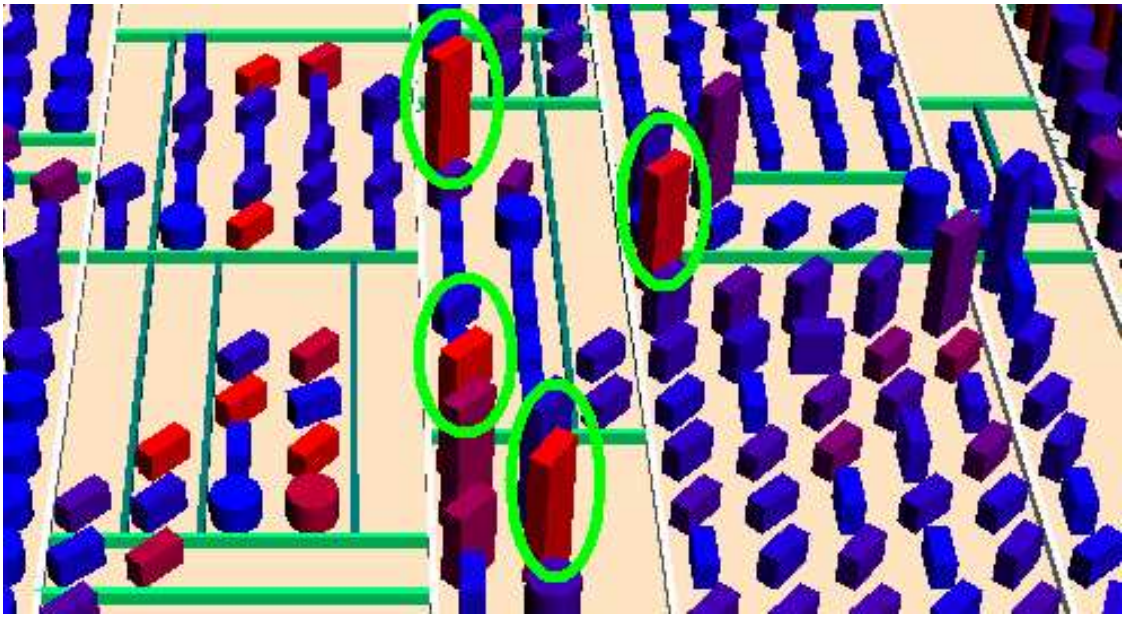


FIG. 5.4 – Quelques exemples de couteau suisse détectés dans *ArgoUML*.

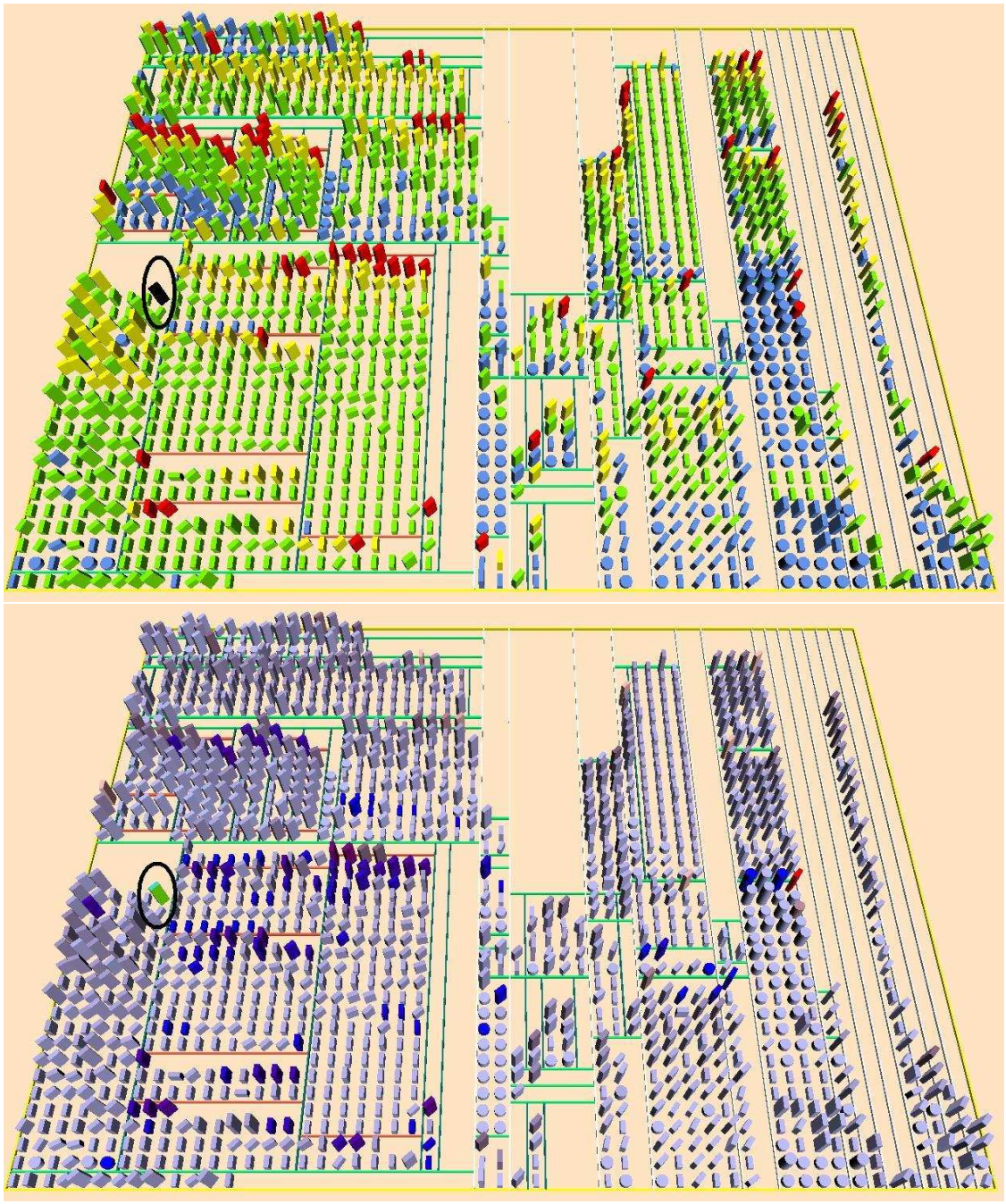


FIG. 5.5 – Un exemple de symptôme de changement divergeant trouvé dans *ArgoUML*. (Haut) Filtre de distribution appliqué sur NUC. (Bas) Filtre d'associations appliqué sur la classe encerclée.

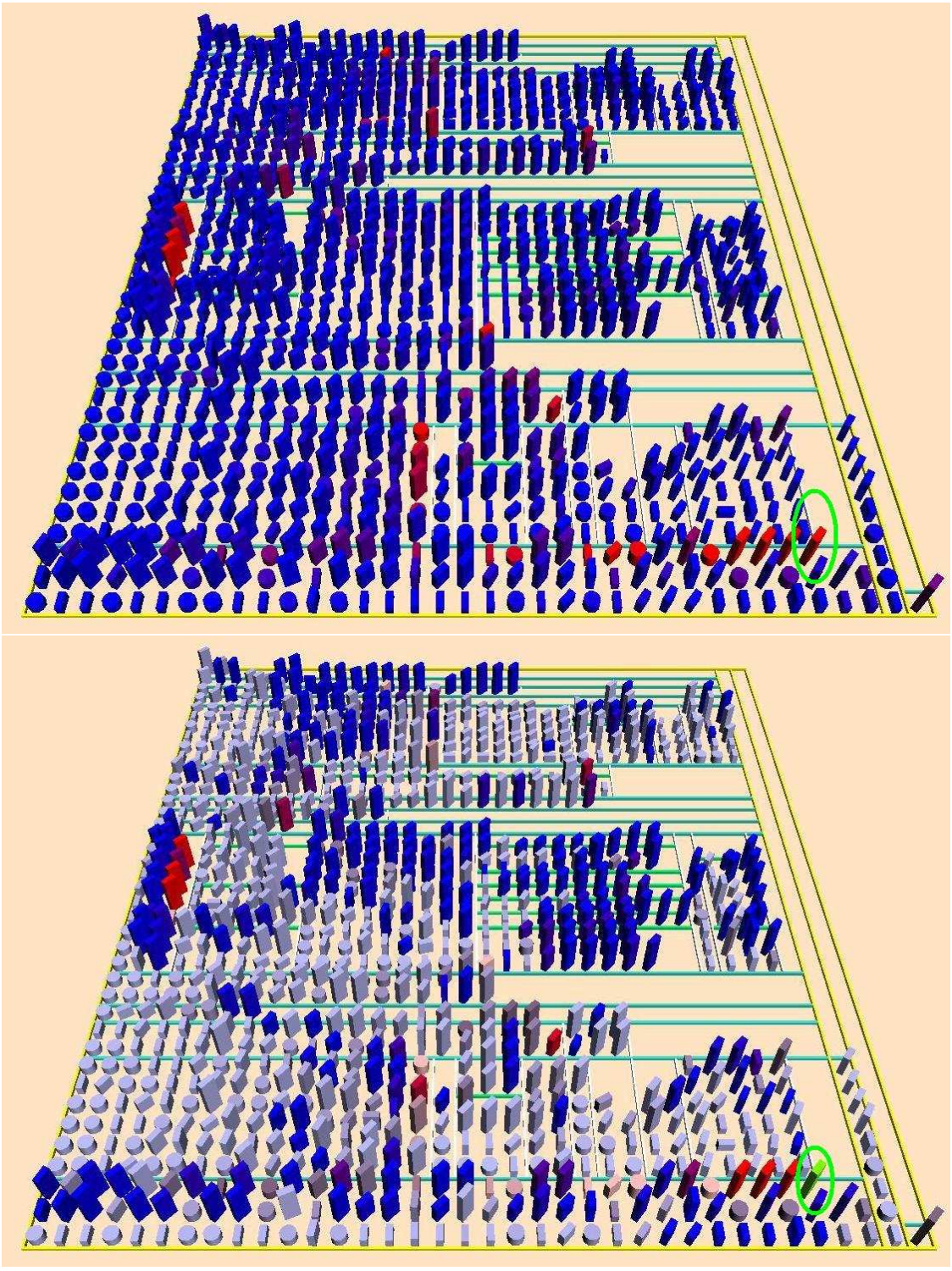


FIG. 5.6 – Un exemple de *shotgun surgery* détecté dans *Freenet*. (Haut) *Mapping* initial pour détecter les candidats au rôle principal. (Bas) Filtre d'associations "entrantes" appliqué sur la classe encerclée.

CHAPITRE 6

VALIDATION

Notre approche de détection d'anomalies est semi-automatique ; l'utilisateur doit prendre des décisions à certains moments lors du processus de détection. Notre objectif est d'offrir un outil de support à la maintenance du logiciel, qui peut guider l'analyste vers des parties du code source où pourraient se trouver des anomalies de conception. Ainsi, le problème d'évaluation de notre approche ne consiste pas à déterminer si l'outil peut accomplir une détection exhaustive comme les outils automatiques, mais plutôt d'évaluer comment il peut aider l'analyste à trouver des anomalies dans un temps raisonnable et de manière efficace.

Pour répondre à cette question, nous avons mené deux études de cas. Dans le reste de ce chapitre, nous présentons le déroulement et les résultats de ces deux études de cas.

6.1 Première étude de cas

6.1.1 Mise en place

Cette étude de cas a été menée avec trois sujets volontaires, des étudiants gradués de notre laboratoire de recherche en génie logiciel. Les sujets avaient une bonne connaissance de la qualité du logiciel (au moins un cours) et de l'expérience de développement et de maintenance de logiciels (au moins un programme de plus de 100 classes).

Les sujets ont utilisé notre approche de détection sur trois logiciels *open-source* connus : *Log4j* 1.2.1 (123 classes), *Lucene* 1.4 (170 classes) et *PMD* 1.8 (286 classes). Pour chaque logiciel, nous avons demandé aux sujets d'identifier les occurrences de quatre anomalies de conception (*blob*, changement divergeant, classe mal placée et *shotgun surgery*) dans un temps limité (20 minutes par programme). Les occur-

rences détectées furent enregistrées dans un fichier par l'outil de détection VERSO. Les résultats ont ensuite été analysés par inspection du code source pour déterminer la proportion des occurrences qui sont des vrais positifs.

6.1.2 Résultats

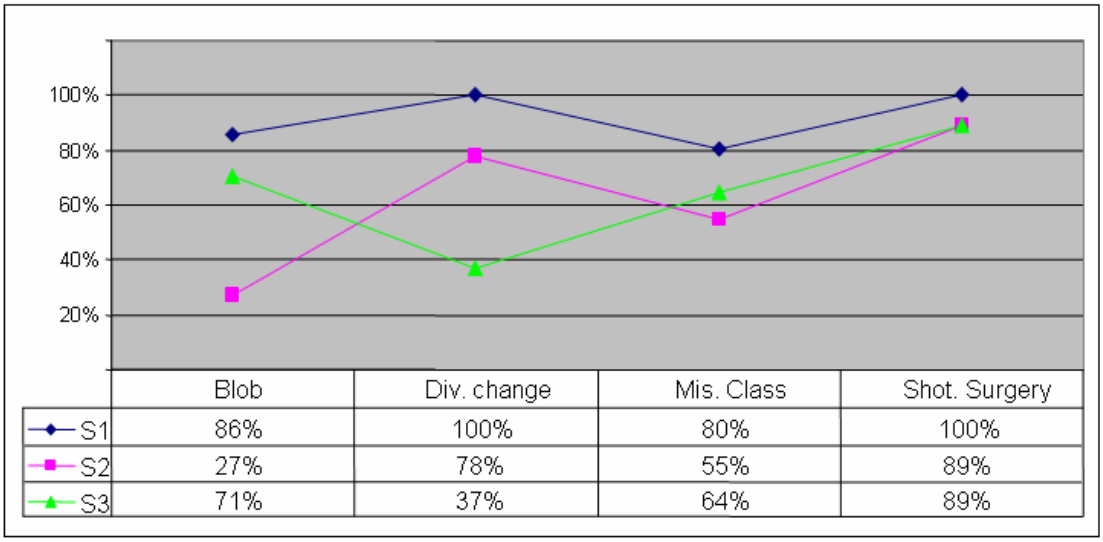
Les résultats de cette étude sont présentés dans le Tableau 6.1. La précision moyenne se situe autour de 60%, à l'exception de *shotgun surgery* qui atteint 92%.

	# Anomalies	# Vrais positifs	Précision
blob	35	21	60%
changement divergeant	32	18	56%
classe mal placée	198	127	64%
shotgun surgery	63	58	92%

TAB. 6.1 – Précision de la détection pour tous les sujets.

En plus de la variabilité entre les types d'anomalies, une analyse approfondie des résultats révèle une variabilité entre les sujets (Figure 6.1) et les programmes (Figure 6.2). En particulier, nous remarquons que les résultats du sujet 1 (le plus expérimenté de tous les sujets) sont élevés pour toutes les anomalies (entre 80% et 100%). Ce sujet a clairement priorisé la précision par rapport à la couverture, si bien qu'il a détecté peu d'occurrences mais avec précision. Le sujet 3 a adopté une stratégie similaire, sauf pour le changement divergeant. L'objectif du sujet 2 semble avoir été de détecter le plus d'anomalies possible (il a détecté plusieurs occurrences pour chaque type d'anomalie) au détriment de la précision.

La variabilité entre les programmes est aussi évidente. À l'exception de *PMD* qui a une très bonne précision pour toutes les anomalies, les autres programmes présentent de bonnes précisions pour deux anomalies seulement, qui ne sont pas les mêmes pour chaque programme. La différence ne peut pas être expliquée par la taille puisque *PMD* est le plus grand des programmes de l'étude.



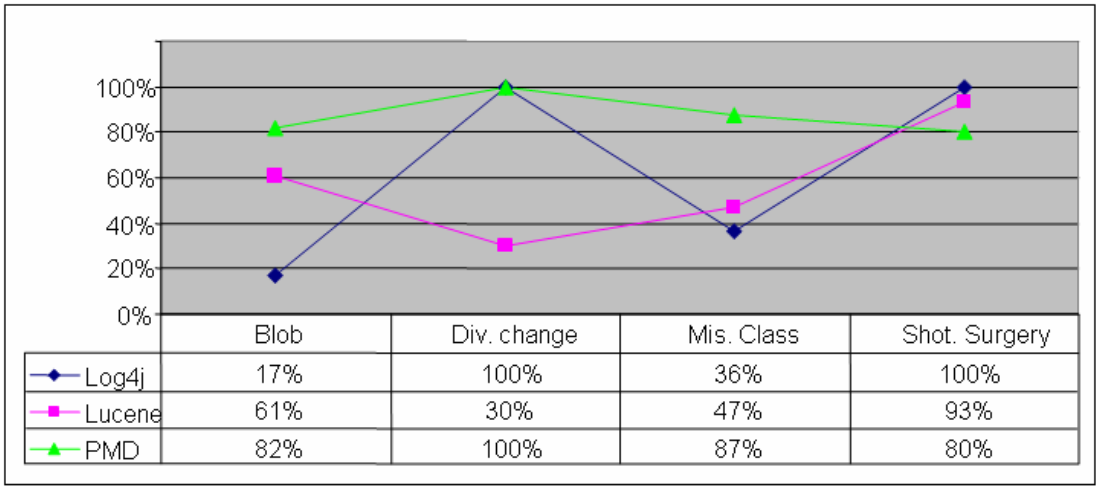
	Blob	Div. change	Mis. Class	Shot. Surgery
S1	86%	100%	80%	100%
S2	27%	78%	55%	89%
S3	71%	37%	64%	89%

FIG. 6.1 – Précisions par sujet.

Bien que les résultats de cette étude de cas ne soient pas généralisables, il est intéressant de discuter des limitations possibles de sa validité [33]. Les sources de biais les plus importantes sont l'hétérogénéité des sujets et le *fishing*. L'hétérogénéité des sujets représente le fait que la variation des résultats due aux différences individuelles pourrait être plus grande que celle due au traitement. Étant donné que notre approche est semi-automatique, les résultats ne seront pas totalement indépendants des compétences et des connaissances des sujets. Il est cependant intéressant d'étudier cette variation afin d'identifier des directions pour améliorer notre approche. L'une d'elles serait d'ajouter des fonctions pour permettre à l'analyste de naviguer plus facilement dans l'espace de recherche.

Le *fishing* signifie que le chercheur désire obtenir un résultat spécifique et pourrait biaiser la conception de l'expérience en conséquence. Dans le cas de notre étude, ceci signifierait que nous aurions choisi des programmes pour lesquels il serait facile de trouver des anomalies de conception. Bien que les programmes utilisés pour l'étude soient de petite taille, ce sont tous des programmes *open source* bien connus dans leur communauté respective.

Au moment du déroulement de cette étude de cas, les extensions de VERSO



	Blob	Div. change	Mis. Class	Shot. Surgery
Log4j	17%	100%	36%	100%
Lucene	61%	30%	47%	93%
PMD	82%	100%	87%	80%

FIG. 6.2 – Précisions par système.

étaient toujours en cours de développement. Par exemple, la fonction de marquage et l'itérateur du filtre de distribution n'étaient pas encore disponibles. Pour la deuxième étude de cas, les sujets ont eu une version complète de VERSO.

6.2 Deuxième étude de cas

6.2.1 Mise en place

Pour cette deuxième étude de cas, 15 sujets ont eu à leur disposition une version de VERSO avec toutes les fonctions de représentation de l'information décrites au Chapitre 4. Cette deuxième étude de cas comprenait un plus grand nombre de sujets que pour la première, ce qui permet de limiter les menaces à la validité de la première étude, comme l'hétérogénéité des sujets.

Les logiciels utilisés sont *PMD 1.8* (286 classes) et *Xerces 1.0.1* (296 classes). Nous avons choisi ces systèmes car ils ont déjà été analysés par une approche de détection automatique pour laquelle les résultats sont disponibles [28]. De plus, ces systèmes sont *open source* et de taille moyenne. Pour cette étude, nous nous sommes intéressés à trois anomalies de conception : *blob*, décomposition fonctionnelle et couteau suisse. Nous avons systématiquement inspecté manuellement les

deux logiciels afin de dresser une liste de toutes les occurrences de ces anomalies. Cette liste est nécessaire pour calculer la couverture.

Les 15 sujets de l'étude étaient des étudiants gradués ayant de l'expérience en programmation et en génie logiciel. Les sujets ont été séparés dans deux groupes A et B. Les sujets du groupe A ont utilisé VERSO sur *PMD* et l'inspection manuelle sur *Xerces* ; les sujets du groupe B ont utilisé l'inspection manuelle sur *PMD* et VERSO sur *Xerces*. Pour effectuer l'inspection manuelle, les sujets avaient à leur disposition l'environnement de développement intégré *Eclipse*. Avec VERSO, les sujets disposaient d'au plus 30 minutes par système pour détecter les occurrences des trois types d'anomalies. Il n'y avait pas de temps minimum ; un sujet pouvait arrêter après moins de 30 minutes s'il croyait que toutes les anomalies avaient été identifiées.

Pour cette deuxième étude de cas, nous avons modifié, par rapport à la première étude, notre interprétation de la précision dans un contexte de détection semi-automatique. Nous considérons plutôt, tel que mentionné au Chapitre 3, qu'un analyste qui utilise VERSO détecte et valide à la fois les occurrences d'anomalies. De plus, les sujets ont reçu comme instruction de seulement marquer les occurrences dont ils étaient absolument certains. Ainsi, toutes les anomalies détectées sont considérées comme de vrais positifs selon l'interprétation de l'analyste. Dans ce contexte, la précision est toujours de 100%. Pour cette raison, nous avons décidé d'exclure la précision des critères d'évaluation. Nous nous sommes plutôt intéressés à la variation du nombre d'occurrences détectées pour les différents sujets et le rappel basé sur la liste d'occurrences que nous avons relevées manuellement pour chaque système.

6.2.2 Résultats

Notre première observation suite à cette étude est relative aux résultats de l'outil de détection automatique. Lorsque nous avons comparé notre liste de rappel avec la liste des anomalies trouvées par l'approche de détection automatique,

nous avons remarqué que certaines classes étaient considérées comme de vraies occurrences d'anomalies par les auteurs de l'outil de détection automatique, mais n'étaient pas considérées comme des anomalies durant notre inspection manuelle. Par exemple, la classe *org.apache.xerces.dom.NodeImpl*, dans le logiciel *Xerces*, était considérée comme une occurrence de *blob* par ces auteurs. Ils ont pris cette décision en considérant le fait que *org.apache.xerces.dom.NodeImpl* possède plusieurs méthodes et attributs et utilise au moins une classe de données. Cependant, suite à l'inspection du code, nous avons conclu que cette classe ne joue pas le rôle de contrôleur dans le système et ne peut donc pas être considérée comme occurrence de *blob* selon nous. Ceci confirme que l'action de décider si un candidat est une anomalie réelle dépend fortement de l'interprétation de l'analyste. Nous avons aussi remarqué que plusieurs anomalies de notre liste n'avaient pas été détectées par l'outil de détection automatique.

Notre deuxième observation est que la variabilité entre les sujets, en considérant le nombre d'occurrences détectées, est moindre avec VERSO qu'avec l'inspection manuelle, bien que les moyennes soient très proches. Le Tableau 6.2 montre l'écart-type du nombre d'occurrences détectées pour l'inspection manuelle et VERSO sur les deux systèmes. Par exemple, l'écart-type du nombre d'occurrences de *blob* détectées dans *PMD* est 2.63 pour l'inspection manuelle et 1.2 pour VERSO. Nous concluons de ces résultats que l'efficacité de la détection d'anomalies en utilisant l'inspection manuelle varie selon les compétences de l'utilisateur, mais cette variation est minimisée lorsque la détection est assistée d'un outil comme VERSO.

	PMD (Manuel / VERSO)	*Xerces* (Manuel / VERSO)
Blob	2.63 / 1.2	3.74 / 2.93
DF	4.56 / 2.55	3.01 / 2.54
CS	4.84 / 0.87	2 / 1.25

TAB. 6.2 – Écart-type pour le nombre d'occurrences détectées.

Finalement, notre troisième observation est liée au rappel des anomalies détectées avec VERSO. Dans notre cas, le rappel est la proportion de toutes les anomalies présentes dans les systèmes, qui ont été détectées avec VERSO, en prenant les résultats des meilleurs sujets. Le Tableau 6.3 montre le rappel pour les trois anomalies avec les deux systèmes. En considérant le temps limité et la taille des programmes, le rappel est très bon pour la décomposition fonctionnelle et moyen pour le *blob*. Ce pourrait être dû au fait que la décomposition fonctionnelle est une anomalie ayant un rôle unique et était probablement plus facile à détecter qu'une micro-architecture de plusieurs classes comme le *blob*. Le rappel pour le couteau suisse est difficile à interpréter car très peu d'occurrences du couteau suisse étaient présentes dans les deux systèmes.

	PMD	*Xerces*
Blob	2/5 (40%)	5/13 (38%)
DF	12/16 (75%)	25/30 (83%)
CS	0/1 (0%)	1/2 (50 %)

Tab. 6.3 – Rappel pour VERSO.

6.3 Conclusion

Dans ce chapitre, nous avons présenté les résultats de deux études de cas au sujet de l'efficacité de notre approche de détection d'anomalies dans des systèmes réels. La première étude mesurait la précision, alors que la deuxième mesurait la variabilité et le rappel. Dans le prochain chapitre, nous résumons le travail présenté dans ce mémoire et présentons des possibilités de travaux futurs.

CHAPITRE 7

CONCLUSION

La détection et la correction d'anomalies est une façon concrète et efficace d'améliorer la qualité du logiciel. Des approches de détection automatique ont été présentées au cours des dernière années. Ces approches génèrent cependant beaucoup de faux positifs et leur précision est très variable en fonction des différentes anomalies à détecter. La détection par inspection manuelle du code source offre la meilleure précision, mais n'est pas réalisable en pratique dans des systèmes réels à cause de son coût en temps et en ressources.

Nous avons proposé une approche semi-automatique de détection qui combine le pré-traitement automatique et la représentation visuelle des données de manière à exploiter les capacités du système visuel humain. Notre approche est complémentaire aux approches automatiques pour les anomalies dont la détection exige une connaissance qui ne peut être extraite directement à partir du code source. Plus précisément, nous détectons des occurrences d'anomalies en les modélisant en tant qu'ensembles de classes jouant des rôles dans des scénarios pré-définis. Des rôles primaires sont identifiés en premier lieu, puis, à partir de ces rôles, des rôles secondaires sont localisés. Cette stratégie nous permet de réduire l'espace de recherche, ce qui permet d'intégrer l'intervention humaine.

Plus concrètement, dans ce mémoire, nous avons présenté notre approche semi-automatique de détection d'anomalies. D'abord, nous avons fait une revue de l'état de l'art dans les domaines de la définition d'anomalies, la détection d'anomalies, ainsi que la visualisation de logiciels. Ensuite, nous avons discuté de problèmes reliés à la détection d'anomalies et présenté les catégories d'informations nécessaires à la détection. Puis, pour chaque catégorie d'information, nous avons expliqué de quelle façon elle est présentée à l'analyste dans VERSO. Nous avons aussi montré les extensions de VERSO réalisées au cours de ce travail pour permettre la détection

d'anomalies. Par la suite, nous avons décrit notre principe général de détection et des stratégies de détection pour six anomalies de conception. Chaque stratégie est accompagnée d'un exemple concret d'occurrence détectée dans des logiciels *open source* de taille représentative dans l'industrie du logiciel. Finalement, nous avons présenté les résultats de deux études de cas réalisées lors de cette maîtrise. La première étude porte sur la précision de la détection. Cette étude a montré qu'un utilisateur qui priorise la précision par rapport au rappel lors de la détection d'anomalies avec VERSO obtient une bonne efficacité. La deuxième étude met l'emphase sur la variabilité des résultats et le rappel. L'étude a montré que la variabilité entre les sujets, en considérant le nombre d'occurrences détectées, est moindre avec VERSO qu'avec l'inspection manuelle. L'étude a également montré que le rappel est généralement bon, compte tenu du temps limité dont les sujets disposaient pour réaliser l'étude.

Comme travaux futurs, nous prévoyons d'abord combiner notre approche avec des approches de détection automatiques. Trois combinaisons sont possibles. Premièrement, nous pouvons utiliser notre outil pour explorer les résultats de la détection automatique et aider à accepter/rejeter les occurrences détectées. Deuxièmement, nous pouvons employer les deux approches en parallèle selon les connaissances requises pour détecter les anomalies. Troisièmement, nous pouvons employer la technique de *focus stylisé* [6] pour attirer l'attention de l'analyste vers les parties du programme où se trouvent les occurrences candidates détectées automatiquement. Ensuite, bien que les résultats des études de cas présentées dans ce mémoire soient intéressants, ils montrent tout de même que l'efficacité de notre approche de détection présente une certaine variabilité par rapport aux compétences de l'utilisateur et selon les types d'anomalies à détecter. Dans le but de réduire cette variabilité au minimum, nous prévoyons développer un assistant pour supporter l'utilisateur lors du processus de détection. Cet assistant pourrait faciliter la navigation et le repérage dans l'espace de recherche. Nous pensons aussi intégrer l'apprentissage machine de sorte qu'après avoir utilisé VERSO pendant un certain

temps, l'assistant pourrait suggérer des occurrences probables d'anomalies à l'utilisateur. Finalement, jusqu'à maintenant, nous utilisons VERSO pour détecter des anomalies en visualisant des métriques logicielles. Nous pensons étendre l'application de notre approche de détection au-delà du domaine du logiciel, à d'autres domaines où l'on s'intéresse également à détecter des phénomènes en utilisant des caractéristiques numériques.

BIBLIOGRAPHIE

[1] El Hachemi Alikacem and Houari A. Sahraoui. Détection d'anomalies utilisant un langage de règle de qualité. In *Actes du* 12^e *Colloque international sur les langages et modèles à objets*, 2006.

[2] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. on Software Engineering*, pages 91–121, 1999.

[3] William J. Brown, Raphael C. Malveau, III Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns : Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[4] Shyam R. Chidamber and Chris F. Kemerer. A metric suite for object-oriented design. *IEEE Trans. on Software Engineering*, 20(6) :293–318, 1994.

[5] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proc. Technology of Object-Oriented Languages and Systems*, pages 18–32, 1999.

[6] Forrester Cole, Doug DeCarlo, Adam Finkelstein, Kenrick Kin, Keith Morley, and Anthony Santella. Directing gaze in 3D models with stylized focus. *Eurographics Symposium on Rendering*, 2006.

[7] Karim Dhambri, Salima Hassaine, Houari Sahraoui, and Pierre Poulin. Détection visuelle d'anomalies de conception. In *Actes du* 14^e *Colloque international sur les langages et modèles à objets*, 2008.

[8] Stéphane Ducasse and Michele Lanza. The class blueprint : Visually supporting the understanding of classes. *IEEE Trans. Software Engineering*, pages 75–90, 2005.

[9] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft – a tool for visualizing line oriented software statistics. *IEEE Trans. on Software Engineering*, 18(11) :957–968, 1992.

[10] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics : A Rigorous and Practical Approach, Revised*. Course Technology, 1998.

[11] James D. Foley, Andries Van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics : Principles and Practice*. Graphics Press, 1990.

[12] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[14] Nathan Gossett and Baoquan Chen. Paint inspired color mixing and compositing for visualization. *IEEE Symposium on Information Visualization*, pages 113–118, 2004.

[15] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships : Putting icing on the UML cake. In *Proc.* 19^{th} *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314, 2004.

[16] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proc. Working Conference on Reverse Engineering*, pages 172–181, 2004.

[17] Salima Hassaine. Utilisation des textures pour la visualisation de logiciels de grande taille. Master's thesis, Université de Montréal, 2007.

[18] Salima Hassaine, Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Automatic generation of strategies for visual anomaly detection. In *Proc.* 14^{th} *ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2007.

[19] Brian Henderson-Sellers. *Object-Oriented Metrics, Measures of Complexity, The Object Oriented Series*. Prentice Hall, 1996.

[20] Johannes Itten. *The Art of Color : The Subjective Experience and Objective Rationale of Color*. Wiley ; Revised edition, 1997.

[21] Guillaume Langelier. Visualisation de la qualité des logiciels de grande taille. Master's thesis, Université de Montréal, 2006.

[22] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proc.* 20^{th} *IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223, 2005.

[23] Michele Lanza and Stéphane Ducasse. Polymetric views – a lightweight visual approach to reverse engineering. *IEEE Trans. on Software Engineering*, 29(9) :782–795, 2003.

[24] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proc. ACM Symposium on Software Visualization*, pages 27–36, 2003.

[25] Radu Marinescu. Detection strategies : Metrics-based rules for detecting design flaws. In *Proc.* 20^{th} *IEEE International Conference on Software Maintenance*, pages 350–359, 2004.

[26] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[27] Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In *Proc.* 21^{th} *IEEE/ACM International Conference on Automated Software Engineering*, pages 297–300, 2006.

[28] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. Discussion on the results of the detection of design defects. In 8^{th} *ECOOP Workshop on Object-Oriented Reengineering*, 2007.

[29] Naouel Moha, Duc-Loc Huynh, and Yann-Gaël Guéhéneuc. Une taxonomie et un métamodèle pour la détection des défauts de conception. In *Actes du 12ième colloque Langages et Modèles à Objets*, pages 201–216, 2006.

[30] V.S. Ramachandran and William Hirstein. The science of art a neurological theory of aesthetic experience. *Journal of Consciousness Studies*, 6 :15–51, 1999.

[31] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[32] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *15th International Conference on Program Comprehension*, pages 231–240, 2007.

[33] Claes Wohlin, Per Runeson, and Martin Höst. *Experimentation in Software Engineering An Introduction*. Springer, 1999.