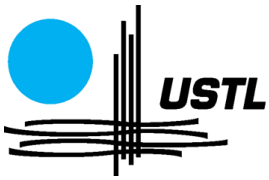
Thèse de Naouel Moha, Lille 1, 2008



École Doctorale régionale Sciences pour l'Ingénieur Lille Nord-de-France

Faculté des arts et des sciences

DECOR : Détection et correction des défauts dans les systèmes orientés objet

THÈSE

en cotutelle

présentée et soutenue publiquement le 26 août 2008 (numéro d'ordre 4206)

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
et
Philosophiæ Doctor (Ph.D.) de l'Université de Montréal

(spécialité informatique)

par

Naouel Moha

Composition du jury

Président :	Mme Esma Aïmeur, *Université de Montréal*
Rapporteurs :	Mme Marianne Huchard, *Université Montpellier II* M. Giuliano Antoniol, *École Polytechnique de Montréal*
Examinateurs :	M. Jean-Marc Jézéquel, *INRIA Rennes* M. Martin Robillard, *McGill University*
Directeurs de thèse :	Mme Laurence Duchien, *Université de Lille I* M. Yann-Gaël Guéhéneuc, *Université de Montréal*
Co-Encadrante :	Mme Anne-Françoise Le Meur, *Université de Lille I*

Département d'informatique et de recherche opérationnelle — Université de Montréal
Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Résumé

Les défauts de code et de conception sont des problèmes d'implémentation et de conception qui proviennent de "mauvais" choix conceptuels récurrents. Ces défauts ont pour conséquence de freiner le développement et la maintenance des systèmes en les rendant plus difficiles à maintenir et évoluer. Une détection et une correction semi-automatiques sont donc des facteurs clefs pour faciliter les phases de maintenance et d'évolution.

Des techniques et outils ont été proposés dans la littérature à la fois pour la détection et la correction des défauts. Les techniques de détection proposées consistent principalement à définir des règles pour détecter les défauts et à les appliquer sur le code source d'un système. Quant aux techniques de correction, elles consistent à appliquer de façon automatique des refactorisations dans le code source du système analysé afin de le restructurer de manière à corriger les défauts. Cependant, la phase qui consiste à identifier les restructurations est réalisée manuellement par les ingénieurs logiciels. Ainsi, il n'est pas possible de corriger directement et automatiquement les défauts détectés. Ce problème est dû au fait que la détection et la correction des défauts sont traitées de façon isolée.

Ainsi, nous proposons DECOR, une méthode qui englobe et définit toutes les étapes nécessaires pour la détection et la correction des défauts de code et de conception. Cette méthode permet de spécifier des règles de détection à un haut niveau d'abstraction et de suggérer des restructurations de code afin d'automatiser la correction des défauts.

Nous appliquons et validons notre méthode sur des systèmes libres orientés objet afin de montrer que notre méthode permet une détection précise et une correction adaptée des défauts.

Mots-clés : défauts de conception, anti-patrons, défauts de code, mauvaises odeurs, spécification, détection, correction, restructurations, refactorisations, JAVA.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Abstract

Title : Detection and Correction of Smells in Object-oriented Systems

Code and design smells are implementation and design problems that come from "poor" recurring design choices. They may hinder development and maintenance of systems by making them hard for software engineers to change and evolve. A semi-automatic detection and correction are thus key factors to ease the maintenance and evolution stages.

Techniques and tools have been proposed in the literature both for the detection and correction of smells. The detection techniques proposed consist mainly in defining rules for detecting smells and applying them to the source code of a system. As for the correction techniques, they consist in applying automatically refactorings in the source code of the system analysed to restructure it and correct the smells. However, software engineers have to identify manually how the system must be restructured. Thus, it is not possible to correct directly and automatically the detected smells. This problem is due to the fact that the detection and the correction of smells are treated independently.

Thus, we propose DECOR, a method that encompasses and defines all steps necessary for the detection and correction of code and design smells. This method allows software engineers to specify detection rules at a high level of abstraction and to obtain automatically suggestions for code restructuring.

We apply and validate our method on open-source object-oriented systems to show that our method allows a precise detection and a suitable correction of smells.

Keywords : design smells, antipatterns, code smells, specification, detection, correction, restructuring, refactorings, JAVA.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Remerciements

La reconnaissance est la mémoire du coeur.
Hans Christian Andersen, écrivain danois (1805-1875).

Je tiens tout d'abord à remercier les membres du jury. Un très grand merci à Marianne Huchard, professeure à l'Université de Montpellier II, et Giuliano Antoniol, professeur associé à l'École Polytechnique de Montréal, d'avoir accepté la fastidieuse tâche de rapporter ce travail. Merci également à Martin Robillard, professeur adjoint à l'Université McGill, et Jean-Marc Jézéquel, professeur à l'Université de Rennes, d'avoir accepté d'examiner mon travail. Enfin, je remercie Esma Aïmeur, professeure à l'Université de Montréal, de m'avoir accordé l'honneur d'être la présidente de mon jury ainsi qu'à Normand Mousseau, professeur agrégé au département de physique de l'Université de Montréal, d'avoir accepté de représenter le doyen de la faculté des études supérieures.

Je remercie également ceux auprès de qui j'ai découvert la recherche : Laurence Duchien, ma directrice de thèse, qui a fait le pari de cette cotutelle et m'a fait profiter de son expérience et de ses compétences scientifiques ; Anne-Françoise, mon encadrante, qui m'a permise de parfaire mon travail en faisant preuve de minutie et de rigueur ; et enfin, Yann-Gaël Guéhéneuc, mon directeur de thèse, pour son soutien, sa disponibilité et ses encouragements. Son dynamisme et son optimisme au quotidien sont, pour moi, un exemple à suivre.

Cette cotutelle m'a permise de côtoyer deux environnements de travail différents et enrichissants : l'équipe projet ADAM à l'Université de Lille I et le laboratoire GEODES de l'Université de Montréal. Je remercie tous mes collègues de part et d'autre de l'Atlantique pour avoir contribué à créer une atmosphère de travail amicale et stimulante. Un merci particulier à Houari Sahraoui, professeur agrégé à l'Université de Montréal, pour ses conseils et ses critiques pertinents sur mon travail, à Petko Valtchev, professeur à l'Université du Québec à Montréal, pour m'avoir fait profiter de ses connaissances et sa rigueur dans l'application de l'analyse formelle de concepts ainsi qu'à Julie Vachon, professeure agrégée à l'Université de Montréal, pour m'avoir aidée dans la compréhension de la vérification formelle. Je remercie également mes collègues et compagnons de fortune pour nos échanges sur mon travail de recherche et pour avoir pris le temps de lire ce document et me faire des commentaires pour l'améliorer ; merci à Amine Mohamed

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Rouane Hacène, El Hachemi Alikacem, Foutse Khomh, May Haydar, Simon Denier et Stéphane Vaucher.

Cette cotutelle de thèse s'est réalisée dans les meilleures conditions grâce aux bourses de recherche et de mobilité de la faculté supérieure des études de l'Université de Montréal, du Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT), du programme de coopération internationale INRIA-FQRNT, de l'ÉGIDE, la direction des relations internationales de l'université de Montréal et du département d'informatique et de recherche opérationnelle de l'Université de Montréal. Je remercie tous ces organismes et institutions pour leur soutien financier.

Je remercie tous mes amis qui m'ont encouragée tout au long de ma thèse dont : Armand, Fathya, Farouk, Jihène, Qing, Mariela, Mirvet, Nadia, Rabia, Rodolphe, Saliha, Zineb et ma très chère et regrettée Souad. Merci à vous tous et à ceux que j'ai oublié pour votre soutien dans les moments difficiles, vos encouragements et votre amitié au quotidien. Une pensée aussi à mon oncle Djilali, ma tante Nourya et mes cousins pour m'avoir chaleureusement accueillie parmi eux.

Merci, enfin, à mes chers parents, mes frères et sœurs, ma petite nièce Chaïma et ma belle-soeur Leïla. Merci pour votre amour et votre soutien inconditionnels.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Préface

CETTE THÈSE a été réalisée dans le cadre d'une cotutelle entre l'Université de Montréal et l'Université des Sciences et Technologies de Lille. Dans le cadre de cette coopération, Naouel Moha a effectué des séjours alternés à raison en moyenne d'une session (4 mois) en France et de deux sessions (8 mois) au Canada sur les 3 ans.

Naouel a été accueillie au sein de l'équipe PTIDEJ dans le laboratoire GEODES à l'Université de Montréal ainsi qu'au sein de l'équipe projet ADAM, INRIA LILLE - NORD EUROPE dans le laboratoire LIFL à l'Université de Lille I.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Note aux lecteurs

NOUS utilisons les conventions typographiques suivantes pour mettre en valeur ou distinguer des éléments du texte et en préciser le sens :

- les noms des langages de programmation, des méthodes, des outils, des produits sont en petites majuscules : JAVA, DECOR ;
- les noms des constituants d'un méta-modèle ou d'un modèle sont en police non-proportionnelle avec empattement : `Figure`, `PolyLineFigure` ;
- les références bibliographiques sont données entre crochets et en français, quelle que soit la langue utilisée pour rédiger le document référencé : [Gamma *et al.*, 1994] ;
- les citations sont entre guillemets et en italique : *"Every model needs a meta model."* [Thomas, 2002, page 18] ;
- les mots importants sont en italique dans la phrase où ils apparaissent ;
- les noms des défauts sont en police sans empattement avec première lettre capitale : Blob, Spaghetti Code ;
- les mots en langue étrangère sont en italique : *refactoring*, *model checking* ;
- nous traduisons en français l'expression anglaise "*software engineers*" par "ingénieurs logiciels".

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Liste des abréviations

AFC	*Analyse Formelle de Concepts*
ARC	*Analyse Relationnelle de Concepts*
BNF	*Backus Normal Form*
COREX	*CORrection EXpert*
DECOR	*DEtection&CORrection*
DETEX	*DETection EXpert*
DSL	*Domain Specific Langage*
DTD	*Document Type Definition*
FCR	*Famille de Contextes Relationnels*
FTR	*Famille de Treillis Relationnels*
IDM	*Ingénierie Dirigée par les Modèles*
PADL	*Pattern and Abstract-level Description Language*
POM	*Primitives, Operators, Metrics*
PTIDEJ	*Pattern Trace Identification, Detection, Enhancement in* JAVA
2DFW	*Design Defects FrameWork*
2DDL	*Design Defects Definition Language*
UML	*Unified Modeling Language*

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Sommaire

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Partie I

Problématique

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Chapitre 1

Introduction

CE CHAPITRE présente le contexte de nos travaux qui s'inscrivent dans la maintenance et l'évolution des systèmes à objets. Nous expliquons nos motivations en montrant que la détection et la correction des défauts sont des activités importantes mais coûteuses dans la maintenance des systèmes. Nous présentons les problèmes associés à la détection et à la correction des défauts dont un problème principal est que la détection et la correction des défauts sont traitées de façon isolée. Enfin, nous présentons le sujet de notre thèse et notre contribution principale, DECOR, une méthode qui permet de détecter et de corriger les défauts dans un même cadre.

Sommaire

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

1.1 Contexte : *la maintenance et l'évolution des systèmes à objets*

À l'issue de leur développement et de leur livraison aux clients, les systèmes logiciels passent en phase de maintenance. Lors de cette phase, les systèmes continuent à être modifiés et adaptés pour supporter les exigences des utilisateurs et les environnements en changement constant ; on parle alors d'évolution.

Cependant, au fil de l'évolution d'un système, sa structure se détériore car les efforts de maintenance se concentrent plus sur la correction des bogues que sur la correction des défauts de conception originaux ou introduits [Frederick P. Brooks, 1975]. En effet, les "mauvaises" solutions à des problèmes communs et récurrents de conception et d'implémentation introduits tôt dans le cycle de développement sont une cause fréquente d'une faible maintenabilité[1] [Klimas *et al.*, 1996] et peuvent freiner l'évolution du système. Il est ainsi difficile pour les ingénieurs logiciels et les mainteneurs d'effectuer des changements tels que l'ajout ou la modification d'une fonctionnalité.

Les mauvaises pratiques de conception sont à l'origine des défauts de conception [Perry et Wolf, 1992]. Ceux-ci sont à l'opposé des patrons de conception [Gamma *et al.*, 1994] et aussi différents des "défauts" tels que définis par Halstead et Fenton, lesquels sont des *"déviations de spécifications ou d'exigences qui peuvent entraîner des défaillances dans les opérations"* [Fenton et Neil, 1999 ; Halstead, 1977]. Les défauts incluent des problèmes de bas niveau tels que les mauvaises odeurs [Fowler, 1999] que nous nommons *défauts de code* et qui sont généralement des symptômes de la présence possible de défauts de haut niveau tels que les anti-patrons [Brown *et al.*, 1998] et désignés comme *défauts de conception*.

Un exemple de défaut de conception bien connu est le Blob [Brown *et al.*, 1998, pages 73–83]. Ce défaut, aussi connu sous le nom de God Class [Riel, 1996], révèle une conception (et une pensée) procédurale implémentée avec un langage de programmation orientée objet. Il se manifeste à travers une large classe contrôleur qui joue un rôle "divin" dans le système en monopolisant le traitement et qui est entourée par un certain nombre de petites classes de données fournissant beaucoup d'attributs mais peu ou pas de méthodes. Dans le Blob, les défauts de code sont la large classe et les classes de données.

Nous utilisons le terme "défauts" dans la suite de la thèse pour désigner à la fois les défauts de code et de conception.

1.2 Motivation : *la réduction des coûts de maintenance*

La nécessité d'adaptation et d'évolution est intrinsèque aux systèmes logiciels. Cependant, selon une des huit lois de Lehman [1996] sur la complexité croissante : *"à mesure qu'un système évolue, sa complexité augmente à moins que des efforts soient fournis pour*

[1] La maintenabilité est la facilité avec laquelle un système logiciel ou un composant peut être modifié afin de corriger des erreurs, d'améliorer la performance ou des autres attributs, ou de s'adapter à un environnement en changement [Geraci, 1991].

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

la réduire et la maintenir". Or, la maintenance logicielle est l'une des activités les plus coûteuses en temps et en ressources dans le processus de développement logiciel [Hanna, 1993 ; Pressman, 2001]. Plusieurs études [Lientz et Swanson, 1980 ; Arthur, 1988 ; Foster, 1993] ont montré que la maintenance logicielle représente de 60% à 80% des dépenses occasionnées tout au long du cycle de développement. Brooks [1975] affirme, quant à lui, que plus de 90% des coûts d'un système sont associés à la phase de maintenance. En l'occurrence, la détection des défauts, qui est une activité réalisée principalement lors de la phase de maintenance, nécessite d'importantes ressources en temps et en personnel et est sujette à beaucoup d'erreurs [Travassos *et al.*, 1999]. Il est donc important de détecter et de corriger au plus tôt les défauts de conception, avant que ces défauts ne puissent être transmis à la prochaine étape du développement ou de la maintenance ou, pire, au client [Travassos *et al.*, 1999]. Plus tôt les défauts sont détectés, plus il est facile de les corriger comme le signale Pressman en citant Niccolo Machiavelli : *"Certaines maladies, comme les médecins disent, à leurs débuts sont faciles à soigner mais difficiles à reconnaître... mais au cours du temps quand elles n'ont pas été reconnues et traitées, deviennent faciles à reconnaître mais difficiles à soigner".*

La détection et la correction des défauts tôt dans le processus de développement peuvent donc réduire considérablement les coûts des activités à venir dans les phases de développement et de maintenance [Pressman, 2001] car des systèmes ou des conceptions dépourvus de défauts sont plus simples à implémenter, changer et maintenir.

1.3 Problème : *l'absence de lien entre détection et correction des défauts*

Suite à l'étude des travaux précédents dans la littérature, nous avons pu observer que la détection et la correction des défauts sont traitées de façon isolée. En effet, les défauts détectés ne peuvent être corrigés directement et automatiquement.

Dans le cadre de ce travail de thèse, nous répondons à ce **problème principal concernant l'absence de lien entre détection et correction des défauts** et nous comblons les lacunes listées ci-dessous.

Lacune 1. Pas de spécifications de haut niveau adaptées aux défauts. Plusieurs approches [Marinescu, 2004 ; Munro, 2005 ; Alikacem et Sahraoui, 2006a] et outils (PMD [2002], RevJava [Florijn, 2002], FindBugs [Hovemeyer et Pugh, 2004]) ont été proposés dans la littérature pour spécifier et détecter les défauts. Cependant, ces approches et outils proposent d'implémenter les règles de détection des défauts en utilisant des langages de programmation. Il est donc difficile pour les ingénieurs logiciels non familiers avec ces langages de spécifier de nouvelles règles selon le contexte du système analysé. Le terme contexte fait référence à l'ensemble des informations précisant les caractéristiques du système analysé, c'est-à-dire le type de système (prototype, système en développement ou en maintenance, système industriel, etc.), les choix de conception (choix associés à

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

des principes ou heuristiques de conception) et les standards de codage (conventions à respecter lors de l'écriture de code source). Les spécifications ont donc besoin d'être adaptées au contexte de chaque système car les caractéristiques d'un système à un autre sont différentes. En effet, les spécifications permettent de définir des caractéristiques propres au système analysé telles que la quantité des commentaires dans le code qui peut être faible dans des prototypes mais élevée dans des systèmes en maintenance, la profondeur d'héritage autorisée qui diffère selon les choix de conception et la taille maximale des classes et des méthodes définie dans les standards de codage.

Lacune 2. Le passage des spécifications des défauts à leur détection est une boîte noire. La plupart des approches n'explicitent pas le passage des spécifications des défauts à leur détection. La façon dont les spécifications sont traitées ou manipulées n'est pas claire et la définition de la plate-forme de détection sous-jacente qui permet de procéder à leur détection n'est pas documentée. Ce manque de transparence empêche toute comparaison, réplication ou amélioration des techniques de détection, et ainsi, de progresser dans le domaine. Ainsi, une connaissance explicite des spécifications des défauts à leur détection permettrait aux développeurs d'outils de comparer leurs techniques de détection avec les outils existants en vue d'améliorer leurs propres outils d'un point de vue de la performance, de la précision et de la couverture des défauts qu'il est possible de détecter.

Lacune 3. La validation des techniques de détection est partielle. Les résultats des détections à partir de ces approches ou outils ne sont pas présentés sur un ensemble représentatif de défauts et de systèmes disponibles. En effet, les résultats disponibles portent sur des systèmes propriétaires et sur un nombre très réduit de défauts. Ainsi, il est difficile de comparer les approches existantes entre elles et de mettre en évidence les forces et faiblesses et donc, les besoins dans ce domaine. Néanmoins, il est important de remarquer qu'une validation précise des techniques de détection est une tâche fastidieuse qui prend du temps.

Lacune 4. La correction des défauts est manuelle. La technique communément utilisée pour corriger les défauts est d'appliquer des refactorisations [Fowler, 1999]. Une refactorisation (en anglais, *refactoring*) est une technique utilisée pour améliorer la structure interne d'un système sans en modifier le comportement externe en effectuant des changements dans le code source [Fowler, 1999]. Ces changements sont communément appelés des restructurations. En effet, selon Arnold [1989], une restructuration (en anglais, *restructuring*) correspond à des modifications apportées au système pour le rendre plus facile à comprendre et à changer ou moins susceptible aux erreurs lorsque des changements futurs sont réalisés.

Les approches proposées dans la littérature pour corriger les défauts consistent à appliquer de façon automatique un ensemble de refactorisations afin de restructurer le système analysé. Cependant, la phase qui consiste à identifier les restructurations à ap-

© 2008 Tous droits réservés.
http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

porter au système analysé n'a pas été étudiée et se fait manuellement. Ainsi, aucune tentative ne suggère comment automatiser la correction des défauts. Comme l'indique Du Bois *et al*. [2004], l'utilisation des refactorisations afin de corriger les défauts et d'améliorer la conception du code n'est pas encore claire et explicite.

Lacune 5. Pas de validation pour la correction des défauts. La validation de la correction des défauts doit être réalisée manuellement par des ingénieurs logiciels et consiste à s'assurer que les corrections, ou plus précisément les restructurations, apportées sur les défauts sont adaptées et pertinentes. Pour le moment, aucune validation n'est disponible dans la littérature sur de telles expérimentations.

1.4 Thèse : *une approche qui fédère détection et correction des défauts*

Notre thèse est qu'il est possible de définir une approche qui fédère détection et correction des défauts et de répondre aux lacunes énoncées ci-dessus. Ainsi, nous proposons la méthode DECOR (*DEtection* & *CORrection*) qui décrit toutes les étapes nécessaires pour la détection et la correction des défauts de code et de conception.

La figure 1.1 (a) présente une vue d'ensemble des cinq étapes de la méthode DECOR liées à la détection des défauts :

- **étape 1. Analyse des descriptions textuelles** : des concepts-clefs sont identifiés dans les descriptions textuelles des défauts de la littérature. Les concepts-clefs désignent des mots-clefs ou des notions spécifiques à la programmation orientée objet utilisés pour décrire les défauts de manière récurrente dans la littérature. Ces concepts-clefs forment un vocabulaire unifié de concepts réutilisables pour décrire les défauts ;
- **étape 2. Spécification** : le vocabulaire de concepts est utilisé pour spécifier de manière cohérente et synthétique les règles de détection des défauts ;
- **étape 3. Traitement** : les spécifications sont traitées et manipulées afin de pouvoir être directement appliquées lors de la détection ; on parle de spécifications opérationnelles ;
- **étape 4. Détection** : la détection est réalisée sur des systèmes en utilisant les spécifications traitées précédemment et retourne la liste des entités de code (classes, méthodes, etc.) suspectes d'avoir des défauts ;
- **étape 5. Validation** : les entités de code suspectes sont validées manuellement pour vérifier si elles ont de réels défauts.

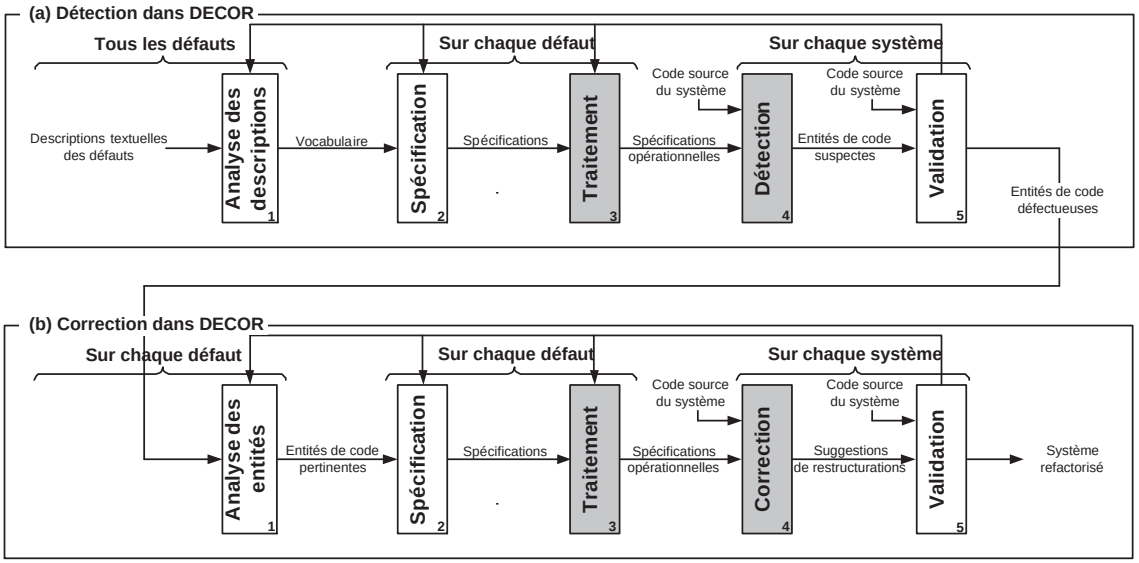
La première étape de la méthode DECOR est générique et est basée sur un ensemble représentatif de défauts. Les étapes 2 et 3 sont suivies lorsqu'un nouveau défaut est spécifié. Les deux dernières étapes 4 et 5 sont répétitives et sont appliquées sur chaque système. Des boucles de retour existent entre les étapes lorsque la validation de la sortie d'une étape suggère de changer l'entrée d'une étape précédente. Lors de cette validation

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Figure 1.1 – La méthode DECOR. (a) La détection et (b) la correction des défauts (*les rectangles représentent des étapes et les flèches connectent les entrées et les sorties de chaque étape. Les rectangles grisés correspondent aux étapes complètement automatiques*).



□

itérative, nous procédons comme suit : dans l'étape 1, nous pouvons enrichir le vocabulaire des défauts ; dans l'étape 2, nous pouvons enrichir le langage de spécification ; dans l'étape 3, les spécifications peuvent nécessiter un traitement plus approprié que le précédent traitement, qui peut avoir mené à des résultats de détection erronés ou imprécis, avant de les ré-appliquer lors de la détection. Par exemple, dans l'étape 3, les spécifications traitées devraient être validées par rapport aux descriptions textuelles. Le critère d'arrêt est défini par les ingénieurs logiciels selon leurs besoins et les sorties de la détection.

Une fois des défauts suspects identifiés, ceux-ci sont traités par la deuxième partie de notre méthode décrit sur la figure 1.1 (b), qui présente les cinq étapes liées à la correction des défauts. Ces étapes suivent le même processus défini au niveau de la détection à la différence qu'à la première étape, ce sont les entités de code suspectes qui sont analysées selon le défaut à corriger :

- **étape 1. Analyse des entités** : cette étape consiste à identifier toutes les entités liées au défaut détecté dans le système analysé ;
- **étape 2. Spécification** : les règles de correction à appliquer sur les entités sont définies sous forme de spécifications ;
- **étape 3. Traitement** : les spécifications sont traitées et manipulées afin de pouvoir être directement appliquées lors de la correction ;

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

- **étape 4. Correction** : la correction est réalisée sur le système analysé en utilisant les spécifications traitées précédemment et retourne la liste des suggestions de restructurations à appliquer ;
- **étape 5. Validation** : les ingénieurs logiciels doivent évaluer les restructurations suggérées et ensuite appliquer les refactorisations permettant de réaliser ces restructurations en utilisant des outils adaptés.

1.5 Contribution : *la méthode* DECOR

L'apport principal de cette thèse est la méthode DECOR dont l'objectif est de traiter de manière cohérente le problème lié à la détection et la correction des défauts, et ainsi, combler le fossé qui existe entre ces deux aspects.

La méthode DECOR, elle-même une contribution, englobe d'autres contributions qui répondent aux lacunes précédemment listées :

Contribution 1. La technique de détection DETEX avec son langage spécifique au domaine. *En réponse à la lacune 1. Pas de spécifications de haut niveau adaptées aux défauts.* Nous avons suivi les étapes de DECOR associées à la détection et implémenté une nouvelle technique de détection, DETEX (*DETection EXpert*), qui permet aux ingénieurs logiciels de spécifier les défauts à un haut niveau d'abstraction en utilisant un vocabulaire unifié et un langage spécifique au domaine. Ce langage, issu d'une analyse de domaine approfondie liée aux défauts, permet de prendre en compte les caractéristiques liées au contexte des systèmes analysés.

Contribution 2. La génération des algorithmes de détection à partir des spécifications est explicite. *En réponse à la lacune 2. Le passage des spécifications des défauts à leur détection est une boîte noire.* Nous présentons en détail le traitement des spécifications et les services fournis par la plate-forme de détection sous-jacente afin d'en permettre la réplication. Dans DETEX, les services fournis par cette plate-forme permettent, à partir des spécifications, de générer directement des algorithmes de détection pour les défauts spécifiés. Ces algorithmes correspondent à du code source JAVA directement exécutable sans aucune intervention manuelle.

Contribution 3. La technique de détection DETEX est validée en terme de précision et de rappel. *En réponse à la lacune 3. La validation des techniques de détection est partielle.* La précision évalue le nombre de défauts réels parmi les défauts détectés et le rappel évalue le nombre de défauts détectés parmi les défauts réels dans le système. Nous validons manuellement DETEX en utilisant le rappel sur le système libre XERCES et la précision sur XERCES et 10 autres systèmes. Ainsi, nous montrons l'utilité et la pertinence de DECOR. Cette validation constitue un premier résultat dans la littérature sur à la fois la précision

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

et le rappel d'une technique de détection sur des systèmes logiciels libres et sur un ensemble représentatif de défauts.

Contribution 4. La technique de correction COREX suggère automatiquement les restructurations à apporter pour corriger les défauts. *En réponse à la lacune 4. La correction des défauts est manuelle.* Tout comme dans la détection, nous avons suivi les étapes de DECOR associées à la correction et implémenté une nouvelle technique de correction, COREX (*CORrection EXpert*), qui suggère automatiquement les restructurations à apporter pour corriger les défauts détectés. Il est important de remarquer que cette technique suggère des restructurations mais n'applique pas concrètement les refactorisations qui mènent à ces restructurations car des outils le permettent déjà de façon automatique ;

Contribution 5. La technique de correction COREX est validée en terme de précision. *En réponse à la lacune 5. Pas de validation pour la correction des défauts.* La précision, ici, évalue le nombre de restructurations pertinentes parmi les restructurations suggérées par l'approche. Tout comme dans la détection, nous validons manuellement notre approche pour la correction sur des instances du défaut de conception Blob détectées dans 4 systèmes libres différents. Notre approche est applicable également sur une dizaine d'autres défauts.

1.6 Plan de la thèse

Ce mémoire de thèse est découpé en trois grandes parties incluant deux chapitres dans chacune des parties et est organisé comme suit :

- la partie I concerne la problématique de la thèse et inclut le chapitre 1 d'introduction déjà présenté ainsi que le chapitre 2 qui présente l'état de l'art sur la détection et la correction des défauts. Ce second chapitre se termine par la mise en perspective de la méthode DECOR avec les travaux de la littérature ;
- la partie II inclut les chapitres 3 et 4 qui décrivent respectivement la détection et la correction des défauts correspondants à l'application de la méthode DECOR ;
- la partie III conclut la thèse en présentant un bilan de nos travaux dans le chapitre 5 et de nos perspectives de recherche dans le chapitre 6.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Chapitre 2

État de l'art sur les défauts, leur détection et leur correction

CE CHAPITRE commence par une description des défauts et, ensuite, présente un état de l'art sur les approches et techniques liées à la détection et la correction des défauts. La plupart des travaux existants se sont focalisés sur la détection *ou* la correction des défauts de façon isolée ; seulement quelques travaux ont abordé les deux aspects. À la fin de ce chapitre, comme bilan, nous mettons la méthode DECOR en perspective par rapport aux travaux existants.

Sommaire

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

2.1 État de l'art sur les défauts

NOUS distinguons deux types de défauts : les *défauts de code* et les *défauts de conception*. Les défauts de code et de conception sont des problèmes communs et récurrents d'implémentation et de conception qui proviennent de "mauvais" choix conceptuels et ont pour conséquence de freiner le développement et la maintenance des systèmes en les rendant plus difficiles à maintenir et évoluer.

Plus précisément, nous utilisons le terme *défauts de conception* pour désigner les défauts de haut niveau et, en particulier, ceux décrits dans la littérature, les anti-patrons [Brown *et al.*, 1998]. Ainsi, les termes *défauts de conception* et *anti-patrons* sont utilisés de manière équivalente sauf que le premier terme est plus général que le second ; ce dernier désigne de manière plus spécifique les défauts de haut niveau décrits dans la littérature [Brown *et al.*, 1998].

De la même façon, nous utilisons le terme *défauts de code* pour désigner les défauts de bas niveau et en particulier ceux décrits dans la littérature, les mauvaises odeurs [Fowler, 1999].

Ainsi, les défauts incluent des problèmes de bas niveau tels que les mauvaises odeurs [Fowler, 1999] que nous nommons *défauts de code* et qui sont généralement des symptômes de la présence possible de défauts de haut niveau tels que les anti-patrons [Brown *et al.*, 1998] et désignés comme *défauts de conception*.

Pour le moment, il n'y a pas d'autres défauts de code et de conception que ceux décrits dans la littérature respectivement comme mauvaises odeurs et anti-patrons. Cependant, les ingénieurs logiciels peuvent décrire leurs propres défauts de code et de conception autres que ceux définis dans la littérature selon leur propre compréhension et le contexte du système analysé.

Un anti-patron [Brown *et al.*, 1998] est une forme littéraire décrivant une mauvaise solution à un problème de conception récurrent qui a un impact négatif sur la qualité de la conception d'un système. Contrairement aux patrons de conception, les anti-patrons décrivent ce qu'il ne faut pas faire. Tout comme les patrons de conception, il existe des anti-patrons généraux [Brown *et al.*, 1998] et des anti-patrons spécifiques à certaines techniques ou sous-domaines du génie logiciel, tels que les processus concurrents [Boroday *et al.*, 2005], J2EE [Dudney *et al.*, 2003 ; Tate et Flowers, 2002], la performance [Smith et Williams, 2002] ou XML [Tate et Flowers, 2002]. Le tableau 2.1 donne la description de quatre anti-patrons décrits dans [Brown *et al.*, 1998] : le Blob [page 73], le Functional Decomposition [page 97], le Spaghetti Code [page 119] et le Swiss Army Knife [page 197]. Le tableau 2.1 résume chacun de ces anti-patrons, lesquels seront à nouveau référencés dans la suite de la thèse.

La méthode `startElement(int, XMLAttrList)` de la classe `org.apache.xerces.validators.dtd.DTDValidator` est un exemple typique de Spaghetti Code. La classe `DTDValidator` scanne une DTD et vérifie que le contenu est valide. Cette classe compte 2 282 lignes de code, déclare 19 méthodes sans paramètres, 12 classes internes, 7 variables de classe, 82 variables d'instance et 138 méthodes d'instance incluant la

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

méthode `startElement`, laquelle a une complexité cyclomatique de McCabe [McCabe, 1976] de 31. Lorsque nous regardons la classe `DTDValidator`, nous pourrions facilement nous exclamer en disant "Berk ! Quel bazar !" [Brown *et al.*, 1998, p. 119]. La description textuelle du Spaghetti Code extrait du livre de [Brown *et al.*, 1998] et un extrait de code de la méthode `startElement` sont donnés dans la figure 2.1.

Beck dans [Fowler, 1999] a décrit 22 mauvaises odeurs, qui correspondent à des structures dans le code source qui suggèrent la possibilité de refactorisations. Nous rappelons qu'une refactorisation (en anglais, *refactoring*) est une technique utilisée pour améliorer la structure interne d'un système sans en modifier le comportement externe en effectuant des changements dans le code source [Fowler, 1999]. Le code dupliqué, les longues méthodes, les larges classes, les classes de données et les longues listes de paramètres sont quelques exemples de défauts de code ou, plus précisément, de mauvaises odeurs. Nous pouvons remarquer dans le tableau 2.1 que la description du Blob inclut les défauts de code, larges classes et classes de données.

Nous avons joint en annexe A, un rapport technique [Tiberghien *et al.*, 2007] correspondant à un répertoire de défauts qui donne la description d'une quarantaine de défauts avec des exemples.

L'utilisation du terme "défaut" n'exclut pas que, dans un contexte particulier, un défaut est la meilleure façon d'implémenter ou de concevoir un système. Par exemple, les analyseurs syntaxiques générés automatiquement sont souvent des Spaghetti Code, c'est-à-dire de larges classes avec de très longues méthodes. Pourtant, bien qu'ils possèdent la structure de défauts, ils n'en sont pas réellement. Ainsi, une intervention manuelle des ingénieurs logiciels est nécessaire afin d'évaluer, selon le contexte, l'occurrence d'un défaut.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Tableau 2.1 – Exemples de défauts de conception.

Le Blob (également appelé God class [Riel, 1996]) correspond à une large classe contrôleur qui dépend de données localisées dans des classes de données associées. Une large classe déclare beaucoup d'attributs et de méthodes et a une faible cohésion. Une classe contrôleur monopolise la plupart du traitement réalisé par le système, prend la plupart des décisions et dirige de près le traitement des autres classes [Wirfs-Brock et McKean, 2002]. Les classes contrôleurs sont identifiables par des noms suspects tels que `Process`, `Control`, `Manage`, `System`, etc. Une classe de données contient seulement des données et réalise peu de traitement sur ces données. Elle est composée d'attributs et de méthodes accesseurs fortement cohésifs.
Le Functional Decomposition peut se produire si des développeurs expérimentés en procédural avec une petite connaissance de l'orienté objet implémentent un système orienté objet. Brown décrit cet anti-patron comme une routine principale qui appelle de nombreuses sous-routines. Le Functional Decomposition correspond à une classe principale avec un nom procédural, tel que `Compute` ou `Display`, dans lequel l'héritage et le polymorphisme sont à peine utilisés. Cette classe est associée à de petites classes, qui déclarent beaucoup d'attributs et implémentent peu de méthodes.
Le Spaghetti Code est un anti-patron qui est caractéristique d'une pensée procédurale dans la programmation orientée objet. Le Spaghetti Code se révèle par des classes sans structure, déclarant de longues méthodes avec pas de paramètres et utilisant des variables de classses. Les noms des classes et méthodes peuvent suggérer de la programmation procédurale. Le Spaghetti Code n'exploite pas et empêche l'utilisation de mécanismes orientés objet tels que le polymorphisme et l'héritage.
Le Swiss Army Knife fait référence à un outil répondant à un large éventail de besoins. L'anti-patron Swiss Army Knife correspond à une classe complexe qui offre un grand nombre de services, par exemple, une classe complexe implémentant un grand nombre d'interfaces. Le Swiss Army Knife est différent du Blob car il expose une grande complexité pour adresser tous les besoins prévisibles d'une partie d'un système, alors que le Blob est un singleton monopolisant tout le traitement et les données d'un système. Ainsi, plusieurs Swiss Army Knives peuvent exister dans un système, par exemple, les classes 'utilitaires'.

□

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Description textuelle

AntiPattern Name : Spaghetti Code
… Anecdotal Evidence : "Ugh ! What a mess !" "You do realize that the language supports more than one function, right ?" "It's easier to rewrite this code than to attempt to modify it." "Software engineers don't write spaghetti code." "The quality of your software structure is an investment for future modification and extension."

…

General Form
Spaghetti Code appears as a program or system that contains very little software structure. Coding and progressive extensions compromise the software structure to such an extent that the structure lacks clarity, even to the original developer, if he or she is away from the software for any length of time. If developed using an object-oriented language, the software may include a small number of objects that contain methods with very large implementations that invoke a single, multistage process flow. Furthermore, the object methods are invoked in a very predictable manner, and there is a negligible degree of dynamic interaction between the objects in the system. The system is very difficult to maintain and extend, and there is no opportunity to reuse the objects and modules in other similar systems.

Symptoms and Consequences

- After code mining, only parts of object and methods seem suitable for reuse. Mining Spaghetti Code can often be a poor return on investment ; this should be taken into account before a decision to mine is made.
- Methods are very process-oriented ; frequently, in fact, objects are named as processes.
- The flow of execution is dictated by object implementation, not by the clients of the objects.
- Minimal relationships exist between objects.
- Many object methods have no parameters, and utilize class or global variables for processing.
- The pattern of use of objects is very predictable.
- Code is difficult to reuse, and when it is, it is often through cloning. In many cases, however, code is never considered for reuse.
- Object-oriented talent from industry is difficult to retain.
- Benefits of object orientation are lost ; inheritance is not used to extend the system ; polymorphism is not used.
- Follow-on maintenance efforts contribute to the problem.
- Software quickly reaches a point of diminishing returns ; the effort involved in maintaining an existing code base is greater than the cost of developing a new solution from the ground up.

…

Extrait de code

Voici un extrait de code de la méthode `startElement` de la classe `org.apache.xerces.validators.dtd.DTDValidator`, un exemple typique de Spaghetti Code.

```
1 public boolean startElement(int, XMLAttrList)
2   throws Exception {
3   ...
4   if (... && !fValidating && !fNamespacesEnabled) {
5       return false;
6   }
7   ...
8   if (contentSpecType == -1 && fValidating) {
9       ...
10  }
11  if (... && elementIndex != -1) {
12      ...
13  }
14  if (DEBUG_PRINT_ATTRIBUTES) {
15      ...
16  }
17  if (fNamespacesEnabled) {
18      fNamespacesScope.increaseDepth();
19      if (attrIndex != -1) {
20          int index = attrList.getFirstAttr(attrIndex);
21          while (index != -1) {
22          ...
23              if (fStringPool.equalNames(...)) {
24                  ...
25              } else {...}
26          }
27      index = attrList.getNextAttr(index);
28          }
29      }
30      int prefix = fStringPool.getPrefixForQName(
31                                              elementType);
32      int elementURI;
33      if (prefix == -1) {
34          ...
35          if (elementURI != -1) {
36              fStringPool.setURIForQName(...);
37          }
38      } else {
39          ...
40          if (elementURI == -1) {
41              ...
42          }
43          fStringPool.setURIForQName(..., elementURI);
44      }
45      if (attrIndex != -1) {
46          int index = attrList.getFirstAttr(attrIndex);
47          while (index != -1) {
48          int attName = attrList.getAttrName(index);
49          if (!fStringPool.equalNames(...)) {
50          ...
51              if (attPrefix != fNamespacesPrefix) {
52              if (attPrefix == -1) {
53                  ...
54              } else {
55                  if (uri == -1) {
56                  ...
57                  }
58                  fStringPool.setURIForQName(attName, uri);
59          ...
60  }
61  if (fElementDepth >= 0) {
62      ...
63  }
64  fElementDepth++;
65  if (fElementDepth == fElementTypeStack.length) {
66      ...
67  }
68  ...
69  return contentSpecType == fCHILDRENSymbol;
70 }
```

FIG. 2.1 – Description textuelle du Spaghetti Code extrait du livre de Brown *et al.* [1998] (*Permission de reproduction de John Wiley & Sons, Inc. pour cet usage seulement*) et extrait de code de la méthode `startElement`.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

2.2 État de l'art sur la détection

DE NOMBREUX TRAVAUX portent sur l'identification de différents types de problèmes dans les systèmes logiciels, les bases de données [Bruno *et al.*, 2007 ; Jorwekar *et al.*, 2007] et les réseaux [Patcha et Park, 2007]. Nous reportons ici seulement les travaux directement liés à la détection des défauts en présentant les descriptions existantes des défauts, les techniques de détection et les outils. La détection des défauts est liée de manière plus générale aux travaux sur la spécification et la détection des patrons et des inconsistances.

2.2.1 Descriptions

Plusieurs livres portent sur les défauts. Webster [1995] a écrit le premier livre sur les défauts dans le contexte de la programmation orientée objet, incluant les pièges conceptuels, politiques, de codage et d'assurance qualité. Riel [1996] a défini 61 heuristiques caractérisant la bonne programmation orientée objet qui traitent des classes, des objets et des relations. Ces heuristiques permettent aux ingénieurs logiciels d'évaluer manuellement la qualité de leurs systèmes et fournissent une base pour en améliorer la conception et l'implémentation. Beck dans [Fowler, 1999] a compilé 22 mauvaises odeurs qui correspondent à des défauts de bas niveau dans le code source des systèmes en suggérant que les ingénieurs doivent appliquer des refactorisations. Les mauvaises odeurs sont décrites dans un style informel et sont accompagnées d'un processus qui permet de les localiser via des inspections manuelles du code source. Brown *et al.* [1998] se sont focalisés sur la conception et l'implémentation des systèmes orientés objet et ont décrit 40 anti-patrons textuellement, lesquels sont des défauts de conception généraux et incluent des anti-patrons bien connus, tels que le Blob et le Spaghetti Code.

Ces livres destinés à une large audience pour un usage éducatif fournissent des vues approfondies sur les heuristiques, les mauvaises odeurs et les anti-patrons. Toutefois, ces livres proposent une inspection manuelle pour identifier des défauts dans le code en se basant seulement sur des descriptions textuelles. Le problème est qu'une telle inspection manuelle du code est une activité sujette à erreurs et coûteuse en temps. Ainsi, certains chercheurs ont proposé des approches de détection automatique des défauts.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

2.2.2 Techniques de détection

Travassos *et al.* [1999] ont proposé un processus bien défini basé sur des inspections manuelles et des *techniques de lecture* pour identifier des défauts. Par contre, aucune tentative n'a été faite pour automatiser ce processus et de ce fait, il ne s'applique pas facilement aux systèmes plus larges. De plus, le processus couvre seulement la détection *manuelle* des défauts.

Marinescu [2004] a présenté une approche basée sur les métriques pour détecter les défauts via des *stratégies de détection*. Cette approche est implémentée dans un outil appelé iPLASMA. Ces stratégies correspondent à des combinaisons de métriques utilisant des mécanismes de composition au travers d'opérateurs ensemblistes et de filtrage via des seuils absolus et relatifs. Plus spécifiquement, ces stratégies capturent des déviations des bons principes de conception. La création d'une stratégie est un processus décomposé en quatre étapes. Tout d'abord, une règle informelle est exprimée sous forme de propriétés de classe. Ensuite, les métriques qui quantifient le mieux chaque propriété sont sélectionnées manuellement. Puis, les valeurs des métriques sont calculées sur un système donné et filtrées selon des seuils associés à chaque propriété. Enfin, les valeurs des métriques sont combinées en utilisant des opérateurs ensemblistes. Bien que Marinescu propose une détection automatique des défauts basée sur un langage de spécification des défauts, il n'explicite pas la plate-forme de détection sous-jacente ainsi que le processus qui permet de passer des spécifications des défauts à leur détection.

Munro [2005] a constaté les limitations des descriptions textuelles et a proposé un patron pour décrire les défauts de manière plus systématique. Ce patron est similaire à celui utilisé dans les patrons de conception [Gamma *et al.*, 1994]. Il consiste en trois parties principales : un nom de défaut, une description textuelle des caractéristiques du défaut, et des heuristiques pour la détection du défaut. Il s'agit d'une étape supplémentaire vers des spécifications plus précises des défauts. Munro a également proposé des heuristiques basées sur les métriques pour détecter des défauts qui sont similaires aux stratégies de détection de Marinescu. De plus, il a réalisé une étude empirique pour justifier le choix des métriques et des seuils pour détecter certains défauts. Munro utilise ses heuristiques pour interpréter les résultats des métriques logicielles sur des systèmes. Par contre, il ne propose pas une technique de détection automatique des défauts.

Alikacem et Sahraoui [2006a ; 2006b] ont proposé un langage de description basé sur des règles pour détecter des défauts et des violations des principes de qualité. Ce langage permet la spécification à un haut niveau d'abstraction de métriques et la spécification de règles utilisant ces métriques. Les règles incluent des informations quantitatives telles que les métriques et des informations structurelles telles que des relations d'héritage et d'association entre classes. Ils utilisent également la logique floue pour exprimer les seuils spécifiés dans les règles. Les règles sont interprétées et exécutées en utilisant un moteur d'inférence. Cependant, cette approche n'a pas été encore validée.

Certaines approches pour des analyses de logiciels complexes utilisent des techniques de visualisation [Dhambri *et al.*, 2008 ; Simon *et al.*, 2001]. De telles approches semi-automatiques sont de bon compromis entre des techniques de détection complètement

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

automatiques qui peuvent être efficaces mais perdent de vue le contexte et une inspection manuelle qui est lente et imprécise [Langelier *et al.*, 2005]. Cependant, ces techniques de visualisation nécessitent une expertise humaine et sont donc coûteuses en temps. D'autres approches [Lanza et Marinescu, 2006 ; van Emden et Moonen, 2002] réalisent une détection complètement automatique des défauts et utilisent des techniques de visualisation pour présenter les résultats de détection. Par exemple, l'outil JCOSMO proposé par Van Emden et Moonen [2002] permet de détecter un ensemble de défauts dans le code source d'un système et de les visualiser sous forme de graphes à l'aide du logiciel de visualisation RIGI [Tilley *et al.*, 1994]. Leur approche consiste à construire une base de faits composés d'informations primitives liées à la structure du système, par exemple "la méthode `m` contient une instruction `switch`" et d'informations dérivées à partir d'informations primitives, par exemple "une class `C` n'utilise aucune des méthodes offertes par ses super-classes". Ces informations sont introduites dans le calculateur d'algèbre relationnelle GROK [Holt, 1998] pour réaliser des opérations de composition, de différence ou de fermeture transitive afin d'inférer des défauts plus complexes qui seront ensuite représentés sous forme de graphes. Cependant, les défauts détectés sont des défauts de code et comprennent uniquement des opérateurs `instanceof`, des transtypages (conversion de types de données) ainsi que la mauvaise odeur Refus d'héritage (en anglais, *Refused Bequest* [Fowler, 1999, page 87]). Leurs efforts ont été poursuivis par Stefan Slinger [2005] dans le cadre d'un mémoire de maîtrise qui consistait à developper un plug-in ECLIPSE, appelé CODENOSE. Ce plug-in reprend la même approche à la différence que les résultats de la détection des défauts sont affichés dans la vue des tâches d'ECLIPSE de la même façon que les erreurs et les avertissements de compilation sont présentés. Il est également possible d'accéder à la localisation de ces défauts dans le code source. La détection des défauts a été complétée par une dizaine de défauts incluant la plupart des mauvaises odeurs définies par Fowler. Cependant, l'ajout de nouveaux défauts nécessite une implémentation. CODENOSE n'offre pas de langage de haut niveau pour spécifier de nouveaux défauts. Le plug-in a été évalué sur le système libre JHOTDRAW.

Tous ces travaux ont contribué de manière significative à la détection automatique des défauts. Cependant, aucun ne propose une solution au problème de la détection qui soit complète et précise incluant à la fois un langage de spécification, une plate-forme de détection explicite, un processus de traitement des spécifications détaillé ainsi qu'une validation de la technique de détection.

2.2.3 Outils

En plus des techniques de détection, plusieurs outils ont été développés pour identifier les défauts, les problèmes d'implémentation et/ou les erreurs de syntaxe.

Dans les années 70, LINT [Johnson, 1977] était un analyseur de programmes C compilés qui signalait les constructions suspectes et les bogues non décelés à la compilation. Le terme *Lint* est utilisé maintenant pour désigner les outils qui réalisent des analyses

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

statiques du code source écrit dans différents langages tels que LCLINT [Evans, 1996], SMALLLINT [Brant, 1997] ou la variante JAVA, JLINT [Artho, 2004].

Des vérificateurs d'annotations tels que ASPECT [1995], LCLINT [Evans, 1996], ou EXTENDED STATIC CHECKER [Detlefs, 1996] utilisent des techniques de vérification de programmes pour identifier des défauts de bas niveau. Ces outils nécessitent l'assistance d'ingénieurs logiciels pour ajouter des annotations dans le code qui peuvent être utilisées pour vérifier l'exactitude du système.

SMALLLINT [Brant, 1997] analyse du code SMALLTALK pour détecter des bogues, des erreurs possibles de programmation ou du code inutilisé. JLINT [Artho, 2004] détecte dans le code JAVA des bogues, des inconsistances et des problèmes de synchronisation en réalisant une analyse de flots de données tout comme FINDBUGS [Hovemeyer et Pugh, 2004] qui détecte des bogues liés à l'exactitude et la performance dans les programmes JAVA. SABER [Reimer *et al.*, 2004] détecte des erreurs de code latentes dans les systèmes J2EE. ANALYST4J [2008] permet l'identification des anti-patrons et des défauts de code dans les programmes JAVA en utilisant des métriques. PMD [2002], CHECKSTYLE [2004], REVJAVA [Florijn, 2002], FXCOP [2006] et les outils commerciaux tels que CODEPRO ANALYTIX [Instantiations, Inc., 2005], JSTYLE [Man Machine Systems, 2005] et KLOCWORK INSIGHT [2006] vérifient la non-conformité à des standards et des styles de codage pré-définis et permettent pour la plupart de calculer des métriques. PMD [2002], SEMMLECODE [2007] et HAMMURAPI [2007] permettent également aux développeurs d'écrire des règles de détection en utilisant JAVA ou XPATH. Cependant, l'ajout de nouvelles règles est destiné aux ingénieurs familiers avec JAVA et XPATH.

CROCOPAT [Beyer *et al.*, 2005] permet de manipuler des relations de n'importe quelle arité avec un langage de manipulation et de requête simple et expressif. Cet outil permet plusieurs analyses structurelles dans des modèles de systèmes orientés objet incluant non seulement la détection des patrons de conception et des problèmes liés au code tels que les cycles mais aussi l'identification de code cloné et de code mort (en anglais, *dead code*).

Un autre groupe d'outils correspond aux vérificateurs de modèles de logiciels (en anglais, *software model checkers*) tels que les outils BLAST [Beyer *et al.*, 2007] et MOPS [Chen et Wagner, 2002]. Ces outils recherchent les violations de propriétés temporelles de sécurité dans les programmes C en utilisant des techniques de vérification de modèles (en anglais, *model checking*).

La plupart de ces outils permettent d'identifier des défauts pré-définis au niveau implémentation tels que des bogues ou des erreurs de codage. Certains comme PMD [2002], SEMMLECODE [2007] et HAMMURAPI [2007] permettent de spécifier de nouveaux défauts en utilisant JAVA ou XPATH. Mais aucun de ces outils ne permet de spécifier de nouveaux défauts avec un langage de haut niveau qui ne nécessite pas de compétences informatiques et qui n'est donc pas accessibles seulement aux informaticiens.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

2.2.4 Autres travaux : détection des patrons et inconsistances

La détection des défauts se rapproche de la détection des patrons de conception. Plusieurs auteurs ont proposé des outils ou approches pour spécifier et identifier des occurrences de micro-architectures similaires à des patrons récurrents, tels que INTENSIVE [Mens *et al.*, 2006], SOUL [Wuyts *et al.*, 1999] ou DEMIMA [Guéhéneuc et Antoniol, 2007]. INTENSIVE est un environnement qui permet la définition, la manipulation et la vérification d'entités du code source structurellement reliées, appelées les *intensional views*. SOUL est un langage déclaratif de méta-programmation basé sur SMALLTALK qui manipule directement les constructions SMALLTALK via des prédicats. DEMIMA est une approche qui utilise la programmation par contraintes basée sur des explications [Jussien et Barichard, 2000] pour détecter les patrons de conception. Cependant, la programmation par contraintes n'est pas adaptée à la détection des défauts contrairement à l'environnement SOUL dont la bibliothèque de prédicats a été utilisée pour identifier et corriger des entités dont les structures et les organisations correspondent à des défauts [Tourwé et Mens, 2003]. D'autres auteurs ont utilisé des requêtes pour identifier les patrons de conception [Kullbach et Winter, 1999] et [Keller *et al.*, 1999]. En particulier, Keller *et al.* [1999] ont proposé l'environnement de rétro-ingénierie SPOOL qui permet l'identification manuelle, semi-automatique ou automatique des composants de conception abstraites en utilisant des requêtes sur des modèles représentant le code source. Les réseaux génériques de raisonnement flou ont également été utilisés dans l'identification des patrons de conception [Jahnke et Zündorf, 1997 ; Niere *et al.*, 2002]. Un patron de conception est décrit sous forme d'un réseau générique de raisonnement flou représentant des règles pour identifier des micro-architectures similaires à son implantation dans le code source. Cependant, cette approche fort prometteuse n'a pas été développée ni implémentée. Enfin, plusieurs auteurs ont proposé des analyses syntaxiques dédiées comme [Brown, 1996], [Hedin, 1997], [Albin-Amiot et Guéhéneuc, 2001] et [Philippow *et al.*, 2005]. Ces analyses sont efficaces en temps, rappel et précision mais sont spécialisées à des patrons de conception particuliers.

D'autres travaux proches incluent des vérificateurs de consistance des architectures [Garlan *et al.*, 1995 ; Allen et Garlan, 1997 ; Dashofy *et al.*, 2005], lesquels ont été intégrés dans des environnements de développement orienté architecture. Par exemple, dans [Dashofy *et al.*, 2005], les critiques, qui représentent des agents actifs, vérifient des propriétés des descriptions d'architecture, identifient des erreurs syntaxiques et sémantiques potentielles et les reportent au concepteur. Les techniques utilisées dans ces travaux sont spécifiques aux architectures.

Ces travaux sont similaires à la détection des défauts dans le sens où il s'agit de détecter une forme particulière dans du code. Cependant, les techniques qu'ils utilisent sont spécifiques à la détection des patrons et des inconsistances et sont difficilement adaptables à la détection des défauts. Néanmoins, la seule technique qui a été inspirée de ces travaux pour la détection des défauts notamment par Tourwé *et al.* [2003] et Alikacem *et al.* [2006a] est le langage de requêtes.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

2.3 État de l'art sur la correction

PEU DE TRAVAUX ont exploré la correction semi- ou complètement automatique des défauts. Les défauts sont souvent encore corrigés en implémentant des analyses de code ou en faisant des transformations manuelles et fastidieuses.

La correction des défauts peut se décomposer en trois étapes principales, éventuellement répétées suite à des essais ou erreurs : (1) identification des modifications pour corriger les défauts ; (2) application des modifications sur le système ; (3) évaluation du système modifié résultant. L'étape (2) de correction a été facilitée par l'introduction récente des *refactorisations* [Fowler, 1999], c'est-à-dire des changements effectués dans le code source d'un système afin d'améliorer sa structure interne sans modifier son comportement externe. Ainsi, des transformations possibles sont maintenant bien comprises et documentées et l'emphase se porte sur l'étape (1) à savoir l'identification des modifications à apporter puisque l'étape (3) concernant l'évaluation des modifications apportées sur le système ne peut être réalisée de manière efficace que manuellement par les développeurs du système ou des ingénieurs logiciels. Nous considérons que l'étape (1) consiste à suggérer des refactorisations ou restructurations à un niveau conceptuel alors que l'étape (2) correspond à l'application concrète de ces refactorisations dans le code source.

Nous décrivons dans la suite les approches proposées pour identifier les modifications à apporter ainsi que les techniques de refactorisation.

2.3.1 Identification des modifications

Trifu *et al.* [2003] ont proposé des stratégies de correction en associant à chaque défaut de conception des solutions possibles. Cependant, une solution est seulement un exemple de la façon dont le système devrait être implémenté afin d'éviter l'apparition d'un défaut plutôt qu'une liste d'étapes que l'ingénieur logiciel doit suivre afin de corriger le défaut.

Sahraoui *et al.* [1999] ont proposé une approche pour identifier des objets dans du code *procédural* ; un problème qui est similaire à la correction d'un Blob mais dans ce cas, le Blob correspond au système dans son ensemble ou à un module de celui-ci. L'approche combine des calculs de métriques avec plusieurs étapes d'analyse basées sur l'analyse formelle de concepts et d'autres raisonnements basés sur les graphes pour détecter des associations parmi les classes nouvellement identifiées. L'Analyse Formelle de Concepts (AFC) [Ganter et Wille, 1999] est une méthode algébrique d'identification de groupes d'individus ayant des propriétés communes. L'approche se décompose en cinq étapes : tout d'abord, certaines métriques telles que le nombre de procédures ou le nombre de variables globales sont calculées pour déterminer le profil du système analysé. Le profil permet de choisir la méthode d'abstraction la plus appropriée pour l'identification d'objets telle que les graphes de référence, les graphes d'interdépendances de procédures ou les graphes des visibilités de types. Ensuite, les objets sont identifiés en utilisant des algorithmes d'AFC de décomposition de graphe et la formation de concepts. La troisième

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

étape consiste à identifier les méthodes de ces objets. Les relations entre objets telles que les associations et les généralisations sont trouvées dans la quatrième étape. Finalement, le programme procédural est transformé en utilisant le modèle objet déterminé.

Snelting et Tip [2000] ont proposé une méthode basée sur l'AFC pour adapter une hiérarchie de classes à un usage spécifique de celle-ci. Elle comprend une étude sur la façon dont les membres de classe (c'est-à-dire les attributs et les méthodes) sont utilisés dans le code client d'un système. L'étude permet l'identification d'anomalies dans la conception des hiérarchies de classes, par exemple, les membres de classe qui sont redondants ou qui peuvent être déplacés vers une classe dérivée. Par contre, en ce qui nous concerne, nous nous focalisons sur des défauts à un niveau d'abstraction plus élevé tels que les défauts de conception. De plus, au-delà de hiérarchies pures, les défauts impliquent des classes associées qui ne sont pas considérées dans cette approche.

Godin et Mili [1993] ont utilisé les treillis de concepts pour la re-conception des hiérarchies de classes en se basant sur les signatures des classes. Pourtant comme Huchard et Leblanc [2000], ils trouvent des restructurations de hiérarchies utiles et la redistribution de membres mais ignorent toute relation entre les éléments d'une classe, propriété importante des défauts. Pour une discussion plus large sur la restructuration des hiérarchies de classes via l'AFC, le lecteur peut également se référer à [Godin et Valtchev, 2005].

Arévalo [2005] a appliqué l'AFC pour identifier des dépendances implicites parmi les classes dans des modèles de systèmes extraits à partir de code source. Un ensemble de vues à différents niveaux d'abstraction sont construits : au niveau classe, les vues montrent l'accès des attributs par les méthodes et les patrons des invocations parmi les méthodes dans une classe, ainsi elles aident à évaluer la cohésion d'une classe. Au niveau d'une hiérarchie de classes, les vues mettent en valeur des dépendances communes et irrégulières au niveau des méthodes et des attributs afin de déduire de possibles restructurations. Au niveau du système, ils ont défini et étendu l'approche de Tonella *et al.* [1999] à des régularités récurrentes telles que les patrons de conception, les contraintes architecturales, les idiomes, etc. Cette approche est similaire et applicable à notre problème de détection des défauts car elle propose des restructurations en se basant sur les éléments d'une classe.

Les travaux de Kirk *et al.* [2006] se rapprochent le plus de notre travail. Ils utilisent une technique de découpage des attributs (en anglais, *attribute slicing*) pour refactoriser les larges classes. Cette technique consiste à "découper" l'ensemble des attributs d'une classe en sous-ensembles en se basant sur l'usage des attributs par les méthodes. L'approche était conçue pour traiter la mauvaise odeur Large Classe et donc, a une étendue d'une unique classe alors que des défauts de conception comme le Blob impliquent de multiples classes. Cependant, cette technique est augmentée d'un découpage intra-méthode qui permet à un ensemble précis d'instructions manipulant un attribut d'être détecté et isolé. Cette approche peut s'appliquer à la détection du Blob mais reste encore à valider.

Simon *et al.* [2001] ont montré que les métriques peuvent servir de support pour l'identification de refactorisations à apporter dans des systèmes orientés objet. Pour cela,

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

ils ont défini une mesure de cohésion qui définit la distance de similarité correspondant au nombre de propriétés partagées par deux entités du système, telles que des méthodes ou des attributs. Leur solution consiste à générer des visualisations en 3D représentant les entités du système et à les afficher selon leurs distances de similarité. Ainsi, les groupes d'entités ayant une petite distance sont fortement cohésifs et inversement ceux avec une grande distance sont faiblement cohésifs. Par contre, l'interprétation des refactorisations doit être réalisée visuellement et nécessite une intervention manuelle.

Bien qu'aucun de ces travaux ne tente de suggérer des refactorisations ou des restructurations pour corriger spécifiquement les défauts, ils adressent des problèmes importants et laissent supposer que l'AFC est une technique prometteuse pour l'identification des modifications à apporter afin de corriger les défauts.

2.3.2 Application des modifications

Les refactorisations ont été introduites en 1992 par Opdyke dans sa thèse de doctorat [Opdyke, 1992]. Les refactorisations peuvent être appliquées au niveau conceptuel, on parle alors de refactorisations composites, ou au niveau code, et dans ce cas, on parle de refactorisations primitives. Les refactorisations composites sont définies en combinant plusieurs refactorisations primitives. Opdyke a défini 26 refactorisations primitives et 3 refactorisations composites ainsi que les préconditions pour assurer la préservation du comportement après leur application. Selon Opdyke, tant que chaque refactorisation primitive préserve le comportement, le résultat de la transformation de la composition préserve également le comportement.

Opdyke a contribué au livre de Martin Fowler [1999], qui décrit plus de refactorisations. Dans l'un des chapitres de ce livre, Beck a également contribué en décrivant comment trouver les mauvaises odeurs dans le code et comment les corriger à l'aide de refactorisations.

Voici trois exemples concrets de refactorisations extraites du livre de Fowler [1999] :

- *Déplacer une méthode* [page 142] consiste à déplacer une méthode `m` d'une classe `A` vers une classe `B`. Ainsi, la méthode `m` sera localisée dans la classe `B` où elle sera plus utilisée que dans la classe `A`.
- *Déplacer un attribut* [page 146] consiste à déplacer un attribut `a` de la classe `A` vers une classe `B` où il sera plus utilisé.
- *Extraire une classe* [page 149] consiste à créer une nouvelle classe `B` et à déplacer tous les attributs et les méthodes pertinents de la classe `A` vers la classe `B`. Cette refactorisation est adaptée dans le cas où une classe a des responsabilités qui devraient être assurées par deux classes distinctes.

Tichelaar *et al.* [2000] ont défini un méta-modèle indépendant du langage, nommé FAMIX, pour représenter les systèmes orientés objet et pour appliquer des refactorisations ; Ils ont également présenté une étude de faisabilité sur les refactorisations primitives pour SMALLTALK et JAVA. FAMIX comprend un ensemble d'entités de code représentant les

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

classes, les méthodes et les attributs. Ce méta-modèle est supporté par l'environnement de ré-ingénierie pour les systèmes orientés objet, MOOSE [Ducasse *et al.*, 2000].

Plusieurs outils permettent d'appliquer des refactorisations de manière simple et efficace tels que :

- XREFACTORY [Xref-Tech, 2000] pour les langages C/C++ intégré dans les environnements EMACS and XEMACS ;
- REFACTORING BROWSER [The Refactory Inc., 1999], un navigateur avancé pour le langage SMALLTALK dans l'environnement VISUALWORKS ;
- JREFACTORY [JRefactory, 2000], REFACTORIT [Aqris, 2002] pour JAVA, extensions dans des environnements tels que JDEVELOPER, NETBEANS, et JBUILDER ;
- et enfin l'environnement ECLIPSE qui contient déjà les refactorisations comme fonctionnalité principale.

Les refactorisations proposées dans ces approches et ces outils peuvent être utilisées afin d'appliquer directement dans le code source les modifications proposées par des approches telles que l'AFC.

2.4 État de l'art sur la détection et la correction

TRÈS PEU DE TRAVAUX ont essayé de détecter *et* de corriger des défauts de manière semi-automatique ou complètement automatique.

Grotehen et Dittrich [1997] ont conçu METHOOD, un processus de conception qui permet aux concepteurs de réviser et d'améliorer continuellement un schéma conceptuel représentant soit une architecture soit un modèle objet ou conceptuel. Ce processus s'appuie sur des heuristiques explicites, des métriques et des règles de transformations. Les heuristiques décrivent comment détecter un défaut de conception, les règles de transformations comment le corriger en générant des fragments de conception alternatifs et les métriques évaluent les caractéristiques des fragments de conception proposés. METHOOD a été implémenté au sein d'un outil appelé MEX (METHOOD *eXpert*) qui offre un éditeur de conception pour éditer un schéma conceptuel dans une vue graphique, une base de données qui stocke le schéma conceptuel conformément à un méta-modèle, ainsi qu'une base de connaissance qui regroupe les mesures, les heuristiques et les règles de transformations sous forme de requêtes OQL (*Object Query Language*). METHOOD permet de détecter et de corriger des défauts conceptuels uniquement lors de la représentation de systèmes durant la phase de conception et donc dépend fortement de la précision des informations fournies par le concepteur au niveau du schéma conceptuel. Cependant, cette approche n'a pas été validée et poursuivie.

Tourwé *et al.* [2003] ont développé un outil pour détecter des défauts et proposer des refactorisations adaptées à leur correction. Cet outil est intégré à VISUALWORKS, un environnement de développement orienté objet pour SMALLTALK, et à REFACTO-

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

RING BROWSER [The Refactory Inc., 1999], un navigateur qui fournit un support pour appliquer des refactorisations. L'outil offre un nombre important de règles de détection, de suggestions de refactorisations et de métriques sous forme de requêtes logiques en SOUL [Wuyts, 1998]. SOUL est un langage proche de PROLOG qui permet de faire des requêtes directement sur des objets SMALLTALK. Cependant, ces règles sont spécifiques à des défauts particuliers et destinées à des ingénieurs logiciels familiers avec PROLOG et SMALLTALK. Il est donc difficile pour d'autres ingénieurs logiciels de les comprendre afin de les réutiliser, de les modifier ou de spécifier de nouvelles règles. Seule une validation partielle de cet outil sur trois petits systèmes d'environ une centaine de classes a été réalisée.

Sahraoui *et al.* [2000] ont exploré l'utilisation des métriques orientées objet comme indicateurs pour détecter automatiquement des situations symptomatiques. Une situation symptomatique est une structure dans le code ou dans la conception d'un système dont les valeurs des métriques indiquent une faible qualité et où une transformation particulière peut être appliquée en vue d'améliorer la qualité de ce système. Les règles de qualité permettant de détecter une situation symptomatique sont fournies par des modèles d'estimation de la qualité. Ces modèles ont été construits à partir de résultats d'études empiriques sur la qualité des systèmes et dérivés via des algorithmes d'apprentissage machine. Plus concrètement, les règles définissent une relation de cause à effet entre des combinaisons de valeurs de métriques et des caractéristiques de qualité telles que la maintenabilité. La détection des situations symptomatiques est basée sur l'analyse de l'impact des métriques sur les divers attributs de qualité. Les transformations suggérées se rapprochent des refactorisations et incluent, par exemple, la création d'une classe abstraite, de sous-classes spécialisées ou d'une classe agrégat. La suggestion des transformations, ou le *principe de prescription* tel que nommé dans l'article, s'appuie sur une relation entre les transformations et des combinaisons prédéfinies de métriques qui définissent de bons attributs de qualité. La prescription est basée sur l'analyse de l'impact des transformations sur les métriques. Le prototype OO1 CORRECTOR développé, mais par la suite abandonné, permettait de calculer plusieurs métriques de qualité dans un système afin de détecter des situations symptomatiques. En se basant sur les estimations des modèles de qualité, l'outil proposait toutes les transformations possibles qui peuvent être appliquées.

Tahvildari et Kontogiannis [2004] ont adopté la même approche que Sahraoui *et al.* [2000] en proposant la plate-forme QDR (*Quality-Driven Reengineering*), qui permet également d'utiliser un ensemble de métriques orientés objet comme indicateurs pour détecter automatiquement des situations où une transformation particulière peut être appliquée en vue d'améliorer la maintenabilité de systèmes orientés objet. Ces deux approches [Sahraoui *et al.*, 2000 ; Tahvildari et Kontogiannis, 2004] sont basées uniquement sur les métriques et ne prennent pas en compte de manière explicite les relations entre classes. Tahvildari et Kontogiannis se focalisent sur les défauts qui ont une forte complexité et un fort couplage alors que Sahraoui *et al.* s'intéressent aux métriques de couplage et d'héritage. Ces restrictions limitent le nombre de défauts qu'il est possible de détecter. De plus, les algorithmes des transformations ne sont pas explicites.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Grant et Cordy [2003] ont proposé RUST (*Refactoring Using Source Transformation*), un environnement Web qui aide le développeur à localiser des défauts en les enveloppant dans le code source par des balises XML et suggérer des refactorisations adaptées. Les règles de détection sont spécifiées en TXL [Cordy *et al.*, 2002], un langage de programmation destiné à l'analyse de logiciels et à la transformation de code source. Cependant, les auteurs n'explicitent pas le type de défauts détectés et aucune experimentation n'a été réalisée pour démontrer l'efficacité de l'approche.

Les travaux présentés sur la détection et la correction des défauts proposent des solutions exploratoires au difficile problème de détecter et de corriger les défauts. Des efforts sont encore nécessaires pour bien définir le problème et proposer une solution satisfaisante et validée.

Bilan

LES APPROCHES et outils présentés dans cet état de l'art offrent des moyens pour spécifier, détecter et corriger des défauts ; cependant, chacune d'eux a ses avantages et se focalise sur un sous-ensemble des étapes nécessaires pour définir de manière systématique une technique de détection et de correction des défauts. De plus, les processus utilisés et les choix faits pour spécifier et implémenter les plates-formes ne sont pas explicités. En effet, ces approches et outils sont souvent dirigés par les services de leur plate-forme de détection ou de correction sous-jacente plutôt que par une étude exhaustive des défauts qui mènent à leurs spécifications, ou plus précisément aux règles de détection et de correction.

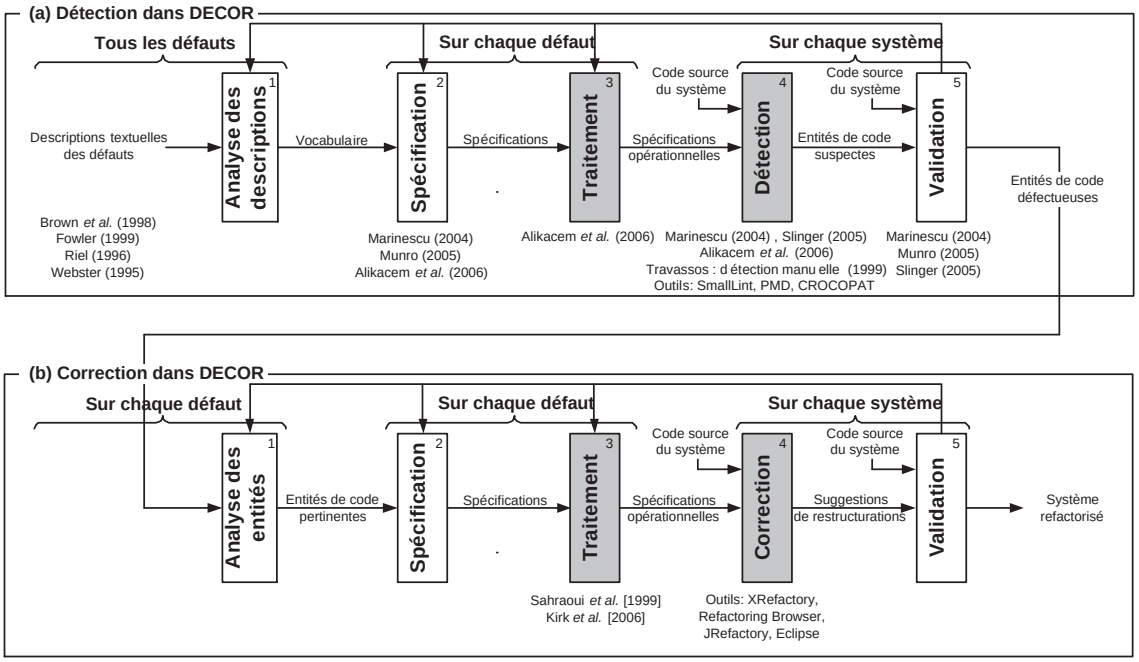
Ainsi, comme contribution principale, nous proposons la méthode DECOR que nous avons présentée dans l'introduction. Les étapes de DECOR sont suivies partiellement par toutes les approches précédentes. Notre méthode regroupe donc les travaux précédents dans un ensemble cohérent et définit explicitement chaque étape ainsi que ses entrées et sorties. DECOR fournit un moyen pour caractériser ces travaux ainsi qu'une base de comparaison pour les travaux futurs. De plus, DECOR est une méthode systématique car elle fournit toutes les étapes nécessaires pour analyser, spécifier, traiter les spécifications, détecter et corriger les défauts, et valider les résultats de la détection et de la correction.

La figure 2.2 met en perspective la méthode DECOR et les travaux précédents. Au niveau de la détection (*cf.* Figure 2.2(a)), certains travaux précédents fournissent des descriptions textuelles des défauts [Brown *et al.*, 1998 ; Fowler, 1999 ; Riel, 1996 ; Webster, 1995] mais aucun ne réalise une analyse complète des descriptions textuelles des défauts. Munro [2005] a amélioré les descriptions des défauts en proposant un patron incluant les heuristiques pour la détection. Cependant, il n'a pas proposé un processus automatique pour leur détection. Marinescu [2004] a proposé une technique de détection basée sur une spécification de haut niveau des défauts. Cependant, il n'a pas explicité le traitement des spécifications, lequel apparaît comme une boîte noire. Alikacem *et al.* [2006a] ont également proposé un langage de spécification de haut niveau et utilisé un moteur

© 2008 Tous droits réservés.
http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 2.2 – La méthode DECOR comparée aux travaux reliés.



□

d'inférence de règles pour traiter les spécifications mais ne fournissent pas de validation de leur approche. Les outils se sont focalisés sur des problèmes d'implémentation et peuvent donc fournir des indications sur les défauts afin d'implémenter des parties de la détection. Bien que ces outils fournissent des langages pour spécifier de nouveaux défauts, ces spécifications sont destinées aux développeurs et donc, ne sont pas des spécifications de haut niveau. Marinescu [2004], Munro [2005] et Slinger [2005] sont les seuls à fournir certains résultats de leur détection mais sur un nombre très réduit de défauts et sur des systèmes propriétaires à l'exception de Slinger [2005], ce qui empêche la réplication de leurs expérimentations.

Au niveau de la correction (*cf.* Figure 2.2(b)), seuls Sahraoui *et al.* [1999] et Kirk *et al.* [2006] explicitent clairement le traitement des spécifications pour la correction des défauts. L'étape de correction c'est-à-dire l'application des restructurations et/ou des refactorisations est assurée par un grand nombre d'outils incluant XREFACTORY [Xref-Tech, 2000], REFACTORING BROWSER [The Refactory Inc., 1999], JREFACTORY [JRefactory, 2000] et ECLIPSE. Ainsi, des efforts sont encore nécessaires pour valider l'approche liée à la correction et en amont définir l'étape qui consiste à analyser les entités issues de la phase de détection et expliciter comment définir les spécifications pour corriger les défauts.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

DECOR est une méthode qui explicite toutes les étapes liées à la détection et la correction des défauts en s'appuyant sur les travaux précédents dans le domaine et permet de valider les résultats de la détection et de la correction.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Deuxième partie

Détection et correction des défauts

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Chapitre 3

Détection des défauts

NOUS avons suivi les étapes de la méthode DECOR pour proposer une nouvelle technique de détection pour les défauts de code et de conception, appelée DETEX (*DETection EXpert*). Nous détaillons l'implémentation de toutes les étapes de DETEX, spécialisées à partir des étapes de DECOR liées à la détection, et plus particulièrement, nous explicitons le traitement qui permet d'aller des spécifications des défauts à leur détection. Ensuite, nous présentons les expérimentations réalisées afin d'évaluer DETEX et ainsi de valider indirectement la méthode DECOR.

Sommaire

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

3.1 DETEX : technique de détection

DETEX est une technique de détection pour les défauts de code et de conception qui suit les étapes définies dans la méthode DECOR.

La figure 3.1 présente une vue d'ensemble des quatre étapes de la technique de détection DETEX, lesquelles ont été spécialisées à partir des étapes génériques de DECOR liées à la détection. Sur la figure, nous mettons en valeur les étapes, les entrées et les sorties qui sont spécifiques à DETEX.

La liste suivante résume les différentes étapes de DETEX :

- **Étape 1. Analyse de domaine** : la première étape consiste à réaliser une analyse approfondie du domaine lié aux défauts pour identifier les concepts-clefs dans les descriptions textuelles des défauts. En plus d'un vocabulaire unifié de concepts réutilisables, une taxonomie et une classification des défauts sont définies en s'appuyant sur les concepts-clefs. Cette taxonomie met en valeur les similarités et les différences entre les défauts.
- **Étape 2. Spécification** : la spécification est réalisée à l'aide d'un langage spécifique au domaine des défauts. Ce langage définit comment spécifier des *fiches de règles* en utilisant le vocabulaire et la taxonomie résultant de l'étape précédente. Une fiche de règles correspond à un ensemble de règles et une règle décrit des propriétés pour détecter les défauts. Le langage permet de définir des propriétés permettant la détection des défauts, de spécifier des relations structurelles entre ces propriétés et caractériser les propriétés selon leur lexique (c'est-à-dire leurs noms), selon leur structure (par exemple, les classes utilisant des variables globales) et selon des attributs internes en utilisant des métriques.
- **Étape 3. Génération d'algorithmes** : les algorithmes de détection sont générés automatiquement à partir des modèles de règles de détection. Ces modèles sont obtenus en réifiant les règles de détection grâce à un analyseur syntaxique et un méta-modèle dédié pour les règles. La réification [Kiczales *et al.*, 1991] consiste à transformer une abstraction, représentée ici par les règles, en un objet concret, ici les modèles des règles. Une plate-forme dédiée supporte la génération automatique des algorithmes de détection. La plate-forme ainsi que le méta-modèle des règles seront présentés en détail dans la section 3.1.3.
- **Étape 4. Détection** : les algorithmes de détection sont appliqués automatiquement sur les modèles des systèmes. Ceux-ci sont obtenus via la rétro-ingénierie (en anglais, *reverse engineering*) de systèmes et correspondent à des instances d'un méta-modèle qui représentent les entités d'un système (classes, interfaces, méthodes, attributs, etc.). Ce méta-modèle est également décrit dans la section 3.1.3.

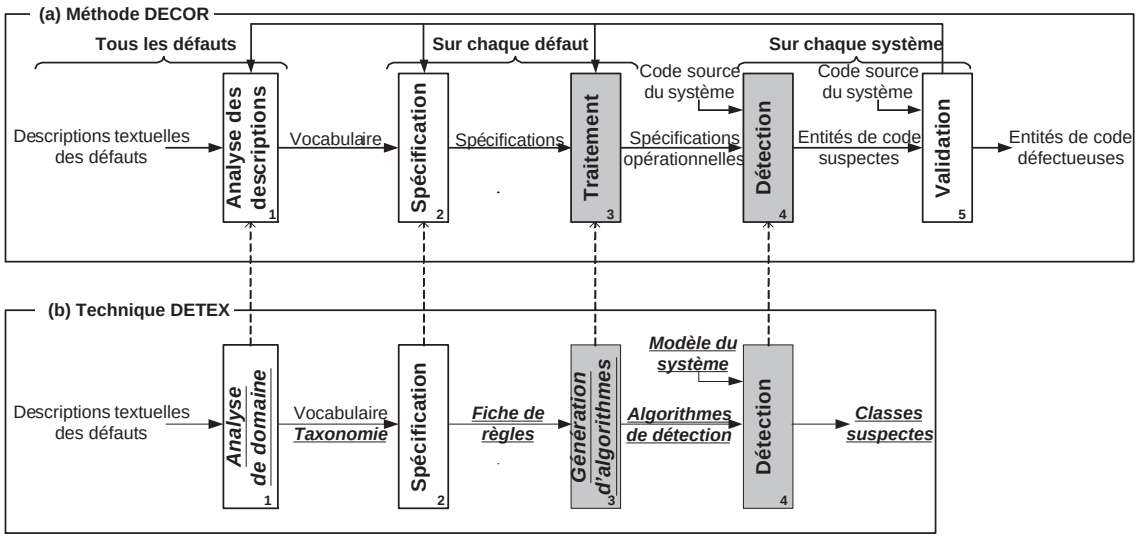
La validation ne fait pas partie de DETEX mais appartient seulement à la méthode DECOR, qui a pour but, à la dernière étape, de valider les résultats d'une telle nouvelle technique de détection.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.1 – (a) La méthode DECOR. (b) La technique de détection DETEX (*les étapes, entrées et sorties en gras, italiques et soulignés sont spécifiques à* DETEX *en comparaison avec* DECOR).



□

La technique de détection DETEX s'inscrit dans une approche IDM (Ingénierie Dirigée par les Modèles). L'IDM [Kent, 2002] est un processus de développement qui utilise des modèles et des transformations de modèles afin de faciliter le développement de systèmes complexes. Ainsi, dans DETEX, à partir de spécifications de haut niveau qui sont représentés ensuite sous forme de modèles, nous aboutissons à des algorithmes de détection directement exécutables suite à un processus de génération, qui s'apparente à des transformations de modèles.

DETEX est originale car les algorithmes de détection ne sont pas ad-hoc mais générés à partir d'un langage spécifique au domaine résultant d'une analyse approfondie de la littérature sur les défauts. Un langage spécifique au domaine offre une plus grande flexibilité que des algorithmes ad-hoc car les experts du domaine, les ingénieurs logiciels ou les experts en qualité logicielle peuvent spécifier et modifier manuellement les règles de détection en utilisant des abstractions de haut niveau liées à leur domaine d'expertise, afin de prendre en compte le contexte des systèmes analysés. Nous rappelons que le contexte du système correspond à l'ensemble des informations précisant les caractéristiques du système analysé, c'est-à-dire le type de système (prototype, système en développement ou en maintenance, système industriel, etc.), les choix de conception (choix associés à des principes ou heuristiques de conception) et les standards de codage (conventions à respecter lors de l'écriture de code source). Le langage permet donc de spécifier des règles de détection adaptées au contexte de chaque système car les caractéristiques d'un système à un autre sont différents. Par exemple, les caractéristiques

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

spécifiques au système analysé incluent la quantité des commentaires dans le code qui peut être faible dans des prototypes mais élevée dans des systèmes en maintenance, la profondeur d'héritage autorisée qui diffère selon les choix de conception et la taille maximale des classes et des méthodes définie dans les standards de codage.

Dans la suite, nous détaillons les quatre étapes de la technique de détection DETEX en utilisant un patron commun : nous présentons le processus de chaque étape incluant les entrées et les sorties de l'étape, la description du processus et son implémentation, ainsi qu'un exemple illustratif, en utilisant le Spaghetti Code, suivi d'une discussion. Nous avons fait le choix d'illustrer DETEX avec le Spaghetti Code car ce défaut permet de bien mettre en valeur l'efficacité de notre technique de détection.

3.1.1 Étape 1 : Analyse de domaine

La première étape de notre technique de détection est inspirée des activités suggérées dans l'analyse de domaine [Prieto-Díaz, 1990].

Selon Prieto-Díaz [1990], l'analyse de domaine est un processus par lequel l'information utilisée pour développer des systèmes logiciels est identifiée, capturée et organisée avec pour but de la rendre réutilisable lors de la création de nouveaux systèmes.

Dans le contexte des défauts, l'*information* se rapporte aux défauts, les *systèmes logiciels* sont les algorithmes de détection et l'information sur les défauts doit être *réutilisable* lors de la spécification de nouveaux défauts.

L'analyse de domaine assure que le langage pour spécifier des défauts est construit au-dessus d'abstractions cohérentes de haut niveau. Elle assure également que ce langage est flexible et expressif.

3.1.1.1 Processus

Nous présentons dans cette section le processus de l'étape d'analyse de domaine incluant l'entrée, la sortie et la description de cette étape. Nous décrivons également comment cette étape a été concrètement implémentée, nous l'illustrons en utilisant le défaut Spaghetti Code et nous terminons par une discussion.

Entrée : les descriptions textuelles des défauts dans la littérature [Brown *et al.*, 1998 ; Fowler, 1999 ; Riel, 1996 ; Webster, 1995].

Sortie : une liste textuelle des concepts-clefs utilisés dans la littérature pour décrire les défauts. Cette liste forme un vocabulaire pour les défauts. Nous utilisons l'analyse de domaine pour fournir une taxonomie des défauts sous la forme d'une carte mettant en valeur les similarités, les différences et les relations entre les défauts. Cette taxonomie a été construite à partir de classifications des défauts de conception et de code.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Description : la première étape concerne l'identification, la définition et l'organisation des concepts-clefs utilisés pour décrire les défauts en incluant des heuristiques basées sur les métriques ainsi que de l'information structurelle et lexicale [Moha *et al.*, 2008a]. Les concepts-clefs désignent des mots-clefs ou des notions spécifiques à la programmation orientée objet utilisés pour décrire les défauts de manière récurrente dans la littérature. Les concepts-clefs forment un vocabulaire de concepts réutilisables pour spécifier les défauts.

L'analyse de domaine nécessite une recherche approfondie dans la littérature pour identifier les concepts-clefs dans les définitions des défauts. Cette analyse doit être réalisée de manière itérative : pour chaque description d'un défaut, les concepts-clefs sont extraits et comparés avec les concepts existants, puis ajoutés au domaine en évitant les synonymes, c'est-à-dire un même concept avec deux noms différents, et les homonymes, c'est-à-dire deux concepts différents avec le même nom. Ainsi, la compilation obtenue de ces concepts forme un vocabulaire concis et unifié.

Les défauts ont été définis et classifiés manuellement en utilisant les concepts-clefs qui les caractérisent. Ces concepts-clefs sont classifiés selon les types de propriétés sur lesquelles ils s'appliquent : propriétés mesurables, lexicales ou structurelles. Les *propriétés mesurables* sont des concepts exprimés avec des mesures d'attributs internes des entités d'un système (classes, interfaces, méthodes, attributs, relations, etc.) telles que la taille des classes ou le nombre de paramètres d'une méthode. Les *propriétés lexicales* correspondent au vocabulaire utilisé pour nommer les entités. Elles caractérisent les entités avec des noms spécifiques définis dans des listes de mots-clefs ou dans un thésaurus. Les *propriétés et les relations structurelles* définissent les structures des entités (par exemple, des attributs correspondant à des variables globales) et leurs relations (par exemple, une relation d'association entre deux classes). Cette classification permet donc d'organiser et de structurer de manière cohérente les défauts à différents niveaux de granularité.

Le vocabulaire est ensuite utilisé pour construire manuellement une taxonomie. Cette taxonomie est construite en intégrant tous les défauts de code et de conception sur une unique carte en identifiant clairement leurs relations. Ainsi, la carte produite organise les défauts, tels que les anti-patrons et les mauvaises odeurs, ainsi que les concepts-clefs reliés, en utilisant les opérateurs ensemblistes tels que l'intersection et l'union. Nous décrivons plus en détail, dans la suite, les classifications des défauts de conception et de code ainsi que la taxonomie sous forme de carte.

Implémentation : cette étape est intrinsèquement manuelle. En effet, elle nécessite des compétences et une expertise des ingénieurs logiciels et peut être difficilement supportée par des outils.

3.1.1.2 Exemple illustratif

Nous résumons la description textuelle du Spaghetti Code [Brown *et al.*, 1998, page 119] dans le tableau 3.1 avec celles du Blob [page 73], du Functional Decomposition

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[page 97] et du Swiss Army Knife [page 197]. Dans la description du Spaghetti Code, nous identifions les concepts-clefs (en italiques dans le tableau) de classes avec des longues méthodes, des noms procéduraux et des méthodes sans paramètres, des classes définissant des variables globales et des classes sans héritage et polymorphisme.

Tableau 3.1 – Liste de défauts de conception (*les concepts-clefs sont en gras et en italiques*).

Le Blob (également appelé God class [Riel, 1996]) correspond à une large classe contrôleur qui ***dépend de données*** localisées dans des classes de données associées. Une large classe déclare ***beaucoup*** d'attributs et de méthodes et a une ***faible*** cohésion. Une classe contrôleur monopolise la plupart du traitement réalisé par le système, prend la plupart des décisions et dirige de près le traitement des autres classes [Wirfs-Brock et McKean, 2002]. Les classes contrôleurs sont identifiables par des noms suspects tels que `Process`, `Control`, `Manage`, `System`, etc. Une classe de données contient seulement des données et réalise peu de traitement sur ces données. Elle est composée d'attributs et de méthodes accesseurs fortement cohésifs.
Le Functional Decomposition peut se produire si des développeurs expérimentés en procédural avec une petite connaissance de l'orienté objet implémentent un système orienté objet. Brown décrit cet anti-patron comme une routine principale qui appelle de nombreuses sous-routines. Le Functional Decomposition correspond à une classe principale avec un nom procédural, tel que `Compute` ou `Display`, dans lequel l'héritage et le polymorphisme sont à peine utilisés. Cette classe est ***associée à de petites classes***, qui déclarent ***beaucoup*** d'attributs et implémentent ***peu*** de méthodes.
Le Spaghetti Code est un anti-patron qui est caractéristique d'une pensée procédurale dans la programmation orientée objet. Le Spaghetti Code se révèle par des classes sans structure, déclarant de ***longues méthodes*** avec ***pas de paramètres*** et utilisant des ***variables de classses***. Les ***noms*** des classes et méthodes peuvent suggérer de la programmation ***procédurale***. Le Spaghetti Code n'exploite pas et empêche l'utilisation de mécanismes orientés objet tels que le ***polymorphisme*** et l'***héritage***.
Le Swiss Army Knife fait référence à un outil répondant à un large éventail de besoins. L'anti-patron Swiss Army Knife correspond à une classe complexe qui offre un ***grand*** nombre de services, par exemple, une classe complexe implémentant un ***grand*** nombre d'interfaces. Le Swiss Army Knife est différent du Blob car il expose une ***grande*** complexité pour adresser tous les besoins prévisibles d'une partie d'un système, alors que le Blob est un singleton monopolisant tout le traitement et les données d'un système. Ainsi, plusieurs Swiss Army Knives peuvent exister dans un système, par exemple, les classes 'utilitaires'.

□

Nous obtenons la classification suivante pour le Spaghetti Code : les propriétés mesurables incluent les concepts de *longues méthodes*, *de méthodes sans paramètres*, d'*héritage* ; les propriétés lexicales incluent les concepts de *noms procéduraux* ; les propriétés structurelles incluent les concepts de *variables globales* et de *polymorphisme*. Les relations structurelles entre constituants apparaissent dans le Blob et le Functional Decomposition, par exemple

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

à travers les concepts-clefs *dépend de données* et *associée à de petites classes*. Les propriétés mesurables sont caractérisées par des valeurs spécifiées en utilisant des mots-clefs tels que *grand, faible, peu* et *beaucoup*, par exemple dans les descriptions textuelles du Blob, du Functional Decomposition et du Swiss Army Knife. Les propriétés peuvent être combinées en utilisant des opérateurs ensemblistes tels que l'intersection ou l'union. Par exemple, toutes les propriétés doivent être présentes pour caractériser une classe comme le Spaghetti Code.

Classification des défauts de code. Beck dans [Fowler, 1999] a fourni un catalogue de mauvaises odeurs mais ne définit aucune classification ou relation entre les mauvaises odeurs. Ce manque de structuration freine l'identification, la comparaison, et par conséquent, la détection des mauvaises odeurs ou des défauts de code.

Des efforts ont été faits pour classifier ces défauts. Mäntylä [2003] a proposé sept catégories, telles que les 'consommateurs' de l'orienté objet (en anglais, *object-orientation abusers*) ou les 'bouffis' (en anglais, *bloaters*) ; cette dernière catégorie inclut les longues méthodes, les larges classes et les longues listes de paramètres. Wake [2003] a distingué les mauvaises odeurs qui se manifestent à l'intérieur d'une classe ou parmi plusieurs classes. De plus, il a distingué les défauts mesurables, les défauts liés à la duplication de code, les défauts dûs à la logique conditionnelle, etc. Ces deux classifications sont basées sur la nature des défauts de code ; cependant, nous nous intéressons à leurs propriétés, leur structure, leur lexique ainsi que leur couverture (intra-classe et inter-classes [Guéhéneuc et Albin-Amiot, 2001]) car ceux-ci reflètent mieux l'étendue des défauts. De plus, un défaut de code peut appartenir à plus d'une catégorie.

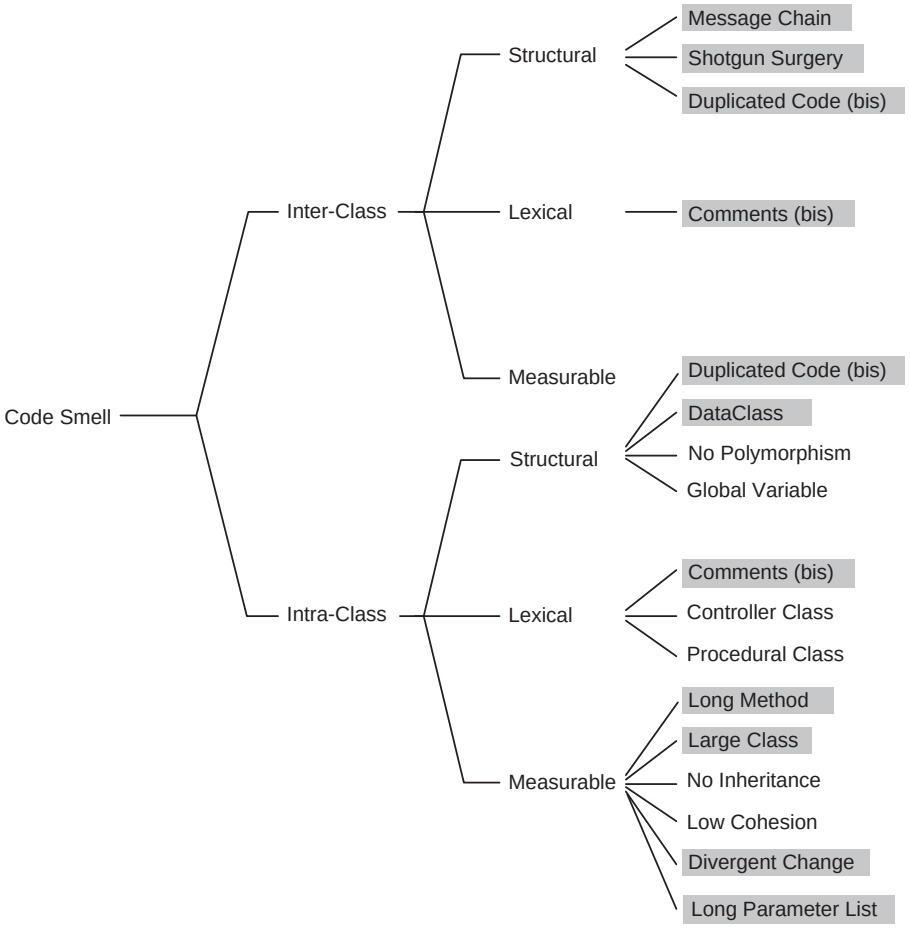
La figure 3.2 montre la classification de certains défauts de code. En accord avec Wake, nous distinguons les défauts de code qui se manifestent à l'intérieur d'une classe et entre plusieurs classes. De plus, nous divisons les deux catégories en : défauts de code *structurels*, liés à la structure d'un système ; *lexicaux*, liés au lexique, qui est exprimé par les noms et verbes utilisés ; et *mesurables*, décrits en utilisant des valeurs de métriques. Cette division supplémentaire aide à identifier les techniques à mettre en œuvre pour détecter ces différents types de défauts. Par exemple, la détection d'un défaut structurel peut être basée essentiellement sur des analyses statiques ; la détection d'un défaut lexical peut reposer sur des traitements de langage naturel ; la détection d'un défaut mesurable peut utiliser des métriques. Notre classification est générique et identifie les défauts de code dans plus d'une catégorie, comme le défaut Duplicated Code.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.2 – Classification de certains défauts de code (*les mauvaises odeurs de Fowler sont surlignées en gris*).



□

Classification des défauts de conception. Brown *et al.* [1998] ont classifié les anti-patrons en trois catégories : développement, architecture et gestion de projet. Nous nous focalisons sur les anti-patrons liés au développement et à l'architecture car ils représentent des mauvaises pratiques de conception. De plus, leur détection est possible semi-automatiquement et leur correction peut améliorer la structure des systèmes.

La figure 3.3 présente la classification des anti-patrons. Nous utilisons la classification des défauts de code pour classifier les anti-patrons selon leurs symptômes. En particulier, nous distinguons entre les défauts *intra-classe*—défauts dans une classe—et les défauts *inter-classes*—défauts affectant plus d'une classe du système. Cette distinction met en valeur l'étendue du système ou de l'inspection de code nécessaire pour détecter un défaut.

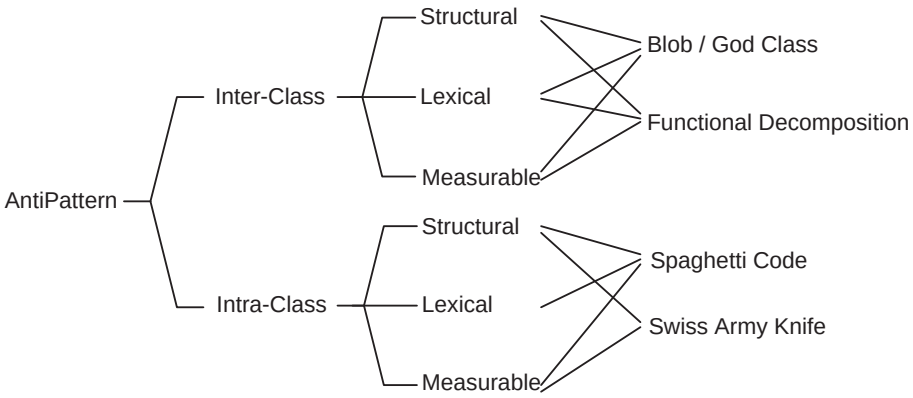
Nous classifions l'anti-patron Spaghetti Code comme un défaut de conception intra-classe appartenant à des sous-catégories structurelles, lexicales et mesurables car ses

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.3 – Classification des anti-patrons.



□

symptômes sont des classes avec des longues méthodes (défaut de code mesurable), définissant des variables globales (défaut de code structurel), avec des noms procéduraux (défaut de code lexical) et n'utilisant pas l'héritage (un autre défaut de code mesurable).

Taxonomie des défauts. La figure 3.4 reprend les classifications comme une taxonomie des défauts et de tous les concepts-clefs reliés sous la forme d'une carte. Celle-ci est similaire à la carte des patrons de Gamma *et al.* [1994, $3^{ième}$ couverture]. Nous incluons seulement les quatre défauts présentés dans le tableau 3.1 dont le Spaghetti Code pour clarification.

Cette taxonomie décrit les relations structurelles parmi les défauts de conception et les défauts de code et les propriétés mesurables, structurelles et lexicales (ovales en blanc). Elle décrit également les relations structurelles (arcs) entre les défauts de conception (hexagones) et les défauts de code (ovales en gris). Elle donne une vue d'ensemble de tous les concepts-clefs qui caractérisent un défaut de conception et les différencie soit comme des relations structurelles entre défauts de code soit comme des propriétés (mesurables, structurelles, lexicales). Elle rend également explicite les relations entre les défauts de conception et de code.

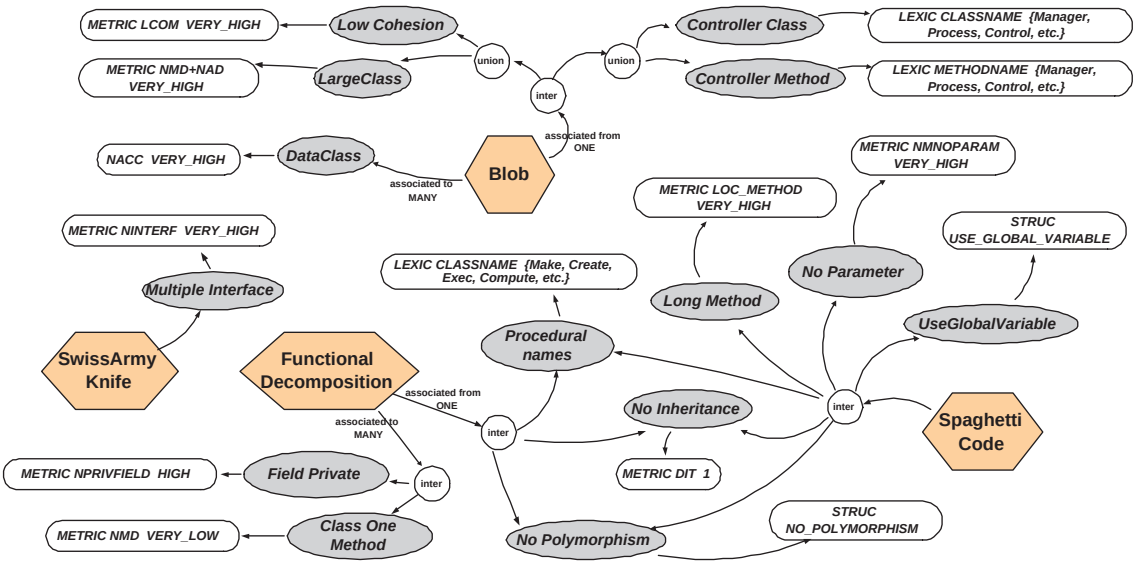
Cette carte est utile pour empêcher toute interprétation erronée de la définition des défauts en clarifiant et classifiant les défauts selon leurs concepts-clefs. Cependant, plusieurs sources d'information peuvent mener à des descriptions contradictoires de défauts ; dans ce cas, le jugement d'un expert du domaine est exigé pour résoudre de telles contradictions. Lanza *et al.* [2006] a introduit la notion de *correlation webs* pour montrer également les relations entre les mauvaises odeurs. Nous avons introduit un niveau supplémentaire d'abstraction en ajoutant des défauts de conception et en incluant plus d'information concernant les propriétés.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.4 – Taxonomie des défauts (*les hexagones correspondent aux anti-patrons, les ovales gris aux mauvaises odeurs et les ovales blancs aux propriétés*).



□

3.1.1.3 Discussion

La distinction entre les défauts *structurels* et *mesurables* n'exclut pas le fait que la structure du système soit mesurable. Cependant, les propriétés structurelles expriment parfois mieux les contraintes entre les classes que les métriques. Alors que les métriques reportent des nombres, nous pouvons vouloir exprimer la présence d'une relation particulière entre deux classes données pour décrire un défaut plus précisément. Les défauts mesurables sont ceux qui peuvent être facilement détectés par une métrique suivant la classification de Wake [2003].

Dans l'exemple du Spaghetti Code, nous utilisons une propriété structurelle pour caractériser le polymorphisme et une propriété mesurable pour l'héritage. Cependant, nous aurions pu utiliser une propriété mesurable pour caractériser le polymorphisme et une propriété structurelle pour l'héritage. De tels choix sont laissés aux experts du domaine qui peuvent choisir la propriété qui correspond le mieux à leur compréhension des défauts et du contexte dans lequel ils veulent les détecter et ce grâce au langage qui permet d'exprimer ces variations. Pour ce qui est des propriétés lexicales, pour le moment, nous utilisons une liste de mots-clefs pour identifier des noms spécifiques mais dans un futur proche, nous comptons utiliser WORDNET, une base de données lexicale en anglais, pour traiter les synonymes afin d'étendre la liste des mots-clefs.

L'analyse de domaine est itérative car l'ajout de la description d'un nouveau défaut peut nécessiter l'extraction d'un nouveau concept, sa comparaison avec les concepts existants et sa classification. Dans le cadre de notre analyse de domaine, nous avons étudié 29

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

défauts incluant 8 anti-patrons et 21 défauts de code. Ces 29 défauts sont représentatifs de l'ensemble tout entier des défauts décrits dans la littérature et incluent plus de 60 concepts-clefs. Ces concepts-clefs, qui sont à différents niveaux d'abstraction et classifiés selon leurs types (mesurable, lexical, structurel), forment un vocabulaire consistant de concepts réutilisables pour spécifier les défauts. Ainsi, notre analyse de domaine est assez complète pour décrire une large variété de défauts et peut être étendue si besoin lors d'une autre analyse de domaine. Par la suite, nous avons pu décrire sans aucune difficulté certains nouveaux défauts qui n'étaient pas utilisés dans l'analyse de domaine. Cependant, cette analyse de domaine ne permet pas la description de défauts liés au comportement du système, comme le Poltergeist [Brown *et al.*, 1998, page 103], mais des travaux en cours [Ng et Guéhéneuc, 2007] nous permettront de décrire et ensuite de spécifier et de détecter cette autre catégorie de défauts.

3.1.2 Étape 2 : Spécification

Dans cette seconde étape, nous présentons, tout comme dans l'analyse de domaine, l'entrée, la sortie, la description et l'implémentation de cette étape. Nous terminons cette section par une illustration avec le Spaghetti Code suivie d'une discussion.

3.1.2.1 Processus

Entrée : le vocabulaire et la taxonomie des défauts.

Sortie : des spécifications détaillant, sous une forme structurée, les règles à appliquer sur le modèle d'un système où détecter les défauts spécifiés.

Description : suite à l'analyse de domaine réalisée dans l'étape précédente, les concepts et les propriétés nécessaires pour décrire les règles de détection à un haut niveau d'abstraction sont formalisés sous la forme d'un langage spécifique au domaine (DSL, acronyme de *Domain Specific Langage*). Le DSL permet de spécifier les défauts de manière déclarative sous forme de compositions de règles dans des *fiches de règles*, en utilisant systématiquement le vocabulaire et la taxonomie des défauts, qui fournissent toute l'information nécessaire sur les défauts et leurs relations pour définir les fiches de règles ; des expérimentations présentées dans la section 3.2 nous permettent d'affirmer cela. Les règles sont associées aux défauts de code et les fiches de règles aux défauts de conception, et plus précisément aux anti-patrons. Ainsi, chaque anti-patron dans la taxonomie correspond à une fiche de règles et chaque défaut de code associé à cet anti-patron dans la taxonomie est décrit comme une règle. Les propriétés dans la taxonomie sont directement utilisées pour exprimer les règles. Le choix d'associer les défauts de code aux règles et les anti-patrons aux fiches de règles permet de faciliter la spécification des défauts sans pour autant manquer de précision. En effet, il est possible de spécifier les défauts directe-

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

ment à l'aide de la taxonomie et du vocabulaire issus de l'analyse de domaine car ceux-ci définissent les propriétés et les relations entre les défauts de conception et de code.

Implémentation : les ingénieurs logiciels définissent manuellement les spécifications des défauts en utilisant le vocabulaire et la taxonomie et, si approprié, en prenant en compte le contexte des systèmes analysés.

Granularité des défauts. Comme indiqué dans la taxonomie, les défauts se rapportent à la structure des classes (en considérant les attributs et les méthodes) ainsi qu'à la structure des systèmes (en considérant les classes et les groupes de classes liées les unes aux autres). Par souci d'uniformité, les défauts caractérisent les classes. Ainsi, une règle détectant des longues méthodes reporte les classes définissant ces méthodes. Une règle impliquant la relation d'association retourne les classes impliquées dans la source de la relation. Il est également possible de retourner les classes cibles de la relation. Ainsi, les règles ont une granularité cohérente et leurs résultats peuvent être combinés en utilisant des opérateurs ensemblistes. Le choix du niveau classe pour le niveau de granularité des défauts ne réduit pas l'utilité de la méthode ou de la technique de détection et a été fait ainsi pour simplifier la spécification des défauts.

3.1.2.2 Langage spécifique au domaine

Le DSL formalise la spécification des fiches de règles avec une grammaire BNF (*Backus Normal Form*). La figure 3.5 montre la grammaire BNF qui définit la syntaxe du DSL. Une fiche de règles est identifiée par le mot clef `RULE_CARD` suivi d'un nom et d'un ensemble de règles spécifiant le défaut (ligne 1). Une règle décrit une liste des propriétés telles que les métriques (lignes 8 à 11), les relations avec les autres règles telles que les associations (lignes 14 à 16) et/ou les combinaisons avec les autres règles basées sur les opérateurs disponibles tels que l'intersection ou l'union (ligne 4). Les propriétés peuvent être de trois types différents : mesurable, structurel ou lexical, et définissent des paires d'identificateur–valeur (lignes 5 à 7).

Nous détaillons dans la suite les différents types de propriétés, d'opérateurs et de relations structurelles qui caractérisent le langage.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.5 – Grammaire BNF des fiches de règles des défauts.

```
 1  rule_card      ::= RULE_CARD:rule_cardName {(rule)+};
 2  rule           ::= RULE:ruleName {content_rule};

 3  content_rule   ::= operator rule rule | property | relationship
 4  operator       ::= INTER | UNION | DIFF | INCL | NEG

 5  property       ::= METRIC id_metric metric_value fuzziness
 6                   | LEXIC id_lexic ((lexic_value,)+)
 7                   | STRUCT id_struct
 8  id_metric      ::= DIT | NINTERF | NMNOPARAM | LCOM | LOC_METHOD | LOC_CLASS
                     | NAD | NMD | NACC | NPRIVFIELD
 9                   | id_metric + id_metric
10                   | id_metric - id_metric
11  value_metric   ::= VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW | NUMBER
12  id_lexic       ::= CLASS_NAME | INTERFACE_NAME | METHOD_NAME | FIELD_NAME | PARAMETER_NAME
13  id_struct      ::= USE_GLOBAL_VARIABLE | NO_POLYMORPHISM | IS_DATACLASS | ABSTRACT_CLASS
                     | ACCESSOR_METHOD | FUNCTION_CLASS | FUNCTION_METHOD | STATIC_METHOD
                     | PROTECTED_METHOD | OVERRIDDEN_METHOD | INHERITED_METHOD | INHERITED_VARIABLE

14  relationship   ::= rel_name FROM rule cardinality TO rule cardinality
15  rel_name       ::= ASSOC | AGGREG | COMPOS
16  cardinality    ::= ONE | MANY | ONE_OR_MANY | OPTIONNALY_ONE

17   rule_cardName, ruleName, lexic_value ∈ string
18   fuzziness ∈ double
```

Propriétés mesurables. Une propriété mesurable définit une valeur numérique ou ordinale pour une métrique donnée (lignes 8 à 11). Les valeurs ordinales sont définies avec une échelle de Likert à 5 points : très élevé, élevé, moyen, faible, très faible. Les valeurs numériques sont utilisées pour définir des seuils alors que les valeurs ordinales sont utilisées pour définir des valeurs relatives par rapport à toutes les classes du système analysé. Les valeurs des métriques peuvent être additionnées ou soustraites. Un degré de logique floue (en anglais, *fuzziness*) correspond à la marge acceptable autour de la valeur numérique ou autour du seuil relatif à la valeur ordinale (ligne 5).

Un ensemble de métriques est identifié durant l'analyse de domaine incluant la suite de métriques de Chidamber et Kemerer [1994] : la profondeur d'héritage `DIT`, le nombre de lignes de code dans une classe `LOC_CLASS`, le nombre de lignes de code dans une méthode `LOC_METHOD`, le nombre d'attributs déclarés dans une classe `NAD`, le nombre de méthodes déclarées dans une classe `NMD`, le manque de cohésion entre les méthodes (en anglais, *Lack of COhesion in Methods*) `LCOM`, le nombre d'accesseurs `NACC`, le nombre d'attributs privés `NPRIVFIELD`, le nombre d'interfaces `NINTERF`, le nombre de méthodes sans paramètres `NMNOPARAM`, etc. Le choix des métriques est basé sur la taxonomie des défauts qui met en valeur les propriétés mesurables nécessaires pour détecter un défaut donné à partir de l'analyse de domaine. Cependant, cet ensemble de métriques n'est pas restreint et peut être facilement étendu avec d'autres métriques.

Nous définissons les valeurs ordinales avec la technique statistique du boxplot (connu également sous le nom de boîte à moustaches) pour associer les valeurs ordinales à des

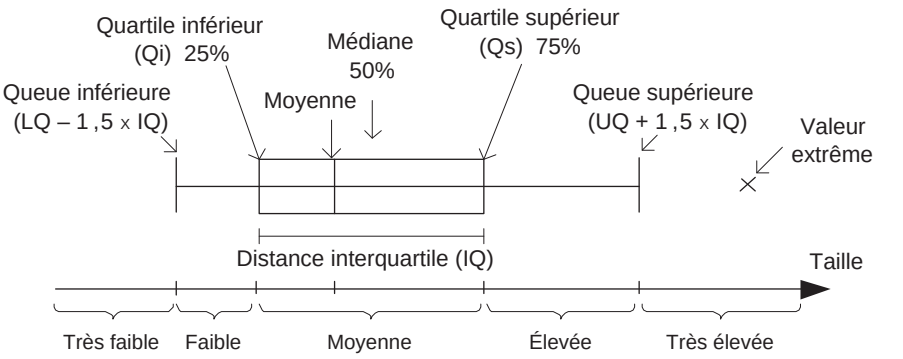
© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

valeurs concrètes de métriques. La figure 3.6 montre un exemple typique de boxplot. Un boxplot [Chambers *et al.*, 1983] partage un ensemble de valeurs numériques en 4 quarts appelés quartiles. Le quartile inférieur Q_i d'un groupe de valeurs est la valeur telle que 25% des valeurs tombent dans ou en-dessous de cette valeur et il représente le $25^{ième}$ percentile. Le quartile supérieur Q_s représente le $75^{ième}$ percentile. Les quartiles inférieurs et supérieurs de la distribution des valeurs représentent les limites du boxplot. La longueur du boxplot ou la distance interquartile, $IQ = Q_s - Q_i$, correspond à la distance entre le quartile inférieur et le quartile supérieur. La distance interquartile est utilisée pour calculer les queues de la distribution qui sont des limites théoriques où toutes les données mesurées devraient se situer si la distribution des données était normale. Les valeurs des queues inférieure et supérieure sont calculées respectivement par $Q_i - 1,5 \times IQ$ et $Q_s + 1,5 \times IQ$. Les valeurs ordinales sont associées aux quartiles comme suit : très faible (très élevé, respectivement) correspond aux valeurs des métriques sous la queue inférieure (au-dessus de la queue supérieure, respectivement) ; faible (élevé, respectivement) correspond aux valeurs entre la queue inférieure et le quartile inférieur (entre la queue supérieure et le quartile supérieur, respectivement) ; moyen correspond aux valeurs entre les quartiles inférieur et supérieur.

FIG. 3.6 – Le boxplot.



□

Le boxplot permet de comparer les valeurs des métriques pour différentes classes et/ou systèmes, sans avoir à fixer de seuils artificiels. Il permet d'identifier également les valeurs extrêmes des métriques. Une valeur est identifiée comme extrême si elle tombe en dehors des queues du boxplot. Cependant, le boxplot ne résout pas complètement le problème des seuils. En effet, si nous inférons que les larges classes sont celles qui ont plus de 40 attributs et méthodes, une classe avec 39 attributs et méthodes ne sera pas considérée comme large. Un degré de logique floue permet de pallier ce problème en spécifiant une marge acceptable autour des seuils identifiés afin d'inclure les valeurs proches de la valeur seuil [Alikacem et Sahraoui, 2006a]. Par exemple, si la distribution des tailles des classes est entre 1 et 50 et que le degré de logique floue est fixé à 10%, nous obtenons une marge de $10\% \times 50 = 5$ et, ainsi, la classe avec 39 attributs et méthodes sera alors considérée comme large avec 10% comme degré de logique floue.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Bien que d'autres outils implémentent la technique du boxplot tels que iPLASMA [Marinescu, 2002], DETEX améliore cette technique avec la logique floue et réduit ainsi le problème lié à la définition des seuils.

Propriétés lexicales. Une propriété lexicale se rapporte au vocabulaire utilisé pour nommer une classe, une interface, une méthode, un attribut ou un paramètre (*cf.* Figure 3.5, ligne 12). Elle caractérise les entités avec des noms spécifiques définis dans une liste de mots-clefs (ligne 6). Dans nos travaux futurs, nous comptons utiliser la base de données de WORDNET pour prendre en compte les synonymes des mots-clefs.

Propriétés structurelles. Une propriété structurelle est une propriété vérifiée sur la structure d'un système (classe, interface, méthode, attribut, paramètre, etc.) (lignes 7 et 13). Par exemple, la propriété `USE_GLOBAL_VARIABLE` est utilisée pour vérifier si une classe utilise des variables globales et la propriété `NO_POLYMORPHISM` est utilisée pour vérifier si une classe qui devrait utiliser le polymorphisme ne le fait pas. La grammaire BNF spécifie seulement un sous-ensemble des propriétés structurelles possibles ; d'autres peuvent être ajoutées lorsque de nouvelles analyses de domaine sont réalisées.

Opérateurs ensemblistes. Les propriétés peuvent être combinées en utilisant des opérateurs ensemblistes incluant l'intersection, l'union, la différence, l'inclusion et la négation (ligne 4). Ici, la négation représente la non-inclusion d'un ensemble dans un autre.

Relations structurelles. Les classes et les interfaces d'un système caractérisées par les précédentes propriétés peuvent être, en plus, reliées les unes aux autres par différents types de relations incluant : l'association, l'agrégation et la composition (lignes 14 à 16). Les cardinalités définissent les nombres minimum et maximum des instances de classes qui participent dans une relation.

3.1.2.3 Exemple illustratif

La figure 3.7 montre la fiche de règles du Spaghetti Code. Cette fiche de règles caractérise des classes comme Spaghetti Code en utilisant l'intersection de deux règles (ligne 2) qui sont elles-mêmes les intersections de deux autres règles (lignes 3 et 6) (dans une prochaine version du DSL, nous envisageons d'intégrer des opérateurs ensemblistes multiples). Une classe est un Spaghetti Code si elle déclare des méthodes avec un très grand nombre de lignes de code (propriété mesurable, ligne 4), sans paramètres (propriété mesurable, ligne 5) ; si elle n'utilise pas l'héritage (propriété mesurable, ligne 8) et le polymorphisme (propriété structurelle, ligne 9), a un nom qui rappelle des noms procéduraux (propriété lexicale, ligne 11) et, enfin, déclare et/ou utilise des variables globales (propriété structurelle, ligne 12).

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.7 – Fiche de règle du Spaghetti Code.

```
1   RULE_CARD: SpaghettiCode  {
2     RULE: Inter0
              {INTER  Inter1  Inter2};
3     RULE: Inter1
              {INTER  LongMethod  NoParameter};
4     RULE: LongMethod          {METRIC LOC_METHOD VERY_HIGH 10.0};
5     RULE: NoParameter         {METRIC NMNOPARAM VERY_HIGH 5.0};
6     RULE: Inter2
              {INTER  Inter3  Inter4};
7     RULE: Inter3
              {INTER  NoInheritance  NoPolymorphism};
8     RULE: NoInheritance       {METRIC DIT 1 0.0};
9     RULE: NoPolymorphism      {STRUCT NO_POLYMORPHISM};
10    RULE: Inter4
              {INTER  ProceduralName  UseGlobalVariable};
11    RULE: ProceduralName      {LEXIC CLASS_NAME  (Make, Create, Exec...)};
12    RULE: UseGlobalVariable   {STRUCT USE_GLOBAL_VARIABLE};
13  };
```

□

3.1.2.4 Discussion

L'analyse de domaine réalisée à l'étape 1 assure que les spécifications sont construites au-dessus d'abstractions cohérentes de haut niveau et qu'elles capturent l'expertise du domaine à la différence des langages à portée plus générale qui sont conçus pour être universels [Consel et Marlet, 1998]. Il est donc plus facile pour les experts du domaine de comprendre les spécifications car elles sont exprimées en utilisant des abstractions liées aux défauts et elles se focalisent sur *quoi* détecter au lieu de *comment* détecter [Consel et Marlet, 1998] comme en programmation logique. Ainsi, le DSL pour spécifier les fiches de règles offre une plus grande flexibilité que d'implémenter des algorithmes de détection ad-hoc. En particulier, nous n'avons fait jusqu'à maintenant aucune référence à l'implémentation concrète de la détection des propriétés et des relations structurelles. En effet, les fiches de règles et les règles peuvent être modifiées facilement par des experts du domaine à un haut niveau d'abstraction sans connaissance de la plate-forme de détection sous-jacente, soit en ajoutant de nouvelles règles soit en modifiant celles existantes pour adapter ces spécifications à différents contextes pour différents systèmes. Les fiches de règles sont utilisées pour spécifier les défauts selon les contextes technologiques et/ou industriels. Par exemple, dans les petits systèmes, un expert du domaine pourra considérer comme défauts des classes avec un `DIT` élevé mais pas dans de larges systèmes. Dans un système de gestion, un expert du domaine pourra également considérer différents mots-clefs comme indiquant des classes contrôleurs. Ainsi, les spécifications sont auto-descriptives et peuvent être modifiées à un haut niveau d'abstraction pour accommoder les spécificités des différents contextes sans aucune connaissance de la plate-forme de détection sous-jacente.

Le langage est concis et expressif et fournit une plate-forme de raisonnement pour spécifier des règles significatives. De plus, nous voulions éviter un langage impératif où, par exemple, si nous avions voulu obtenir les classes qui ont des méthodes sans pa-

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

ramètres, nous aurions spécifié une règle de la forme : `method[1].parameters.size = 0`. En effet, nous avons voulu que notre langage ne nécessite pas des compétences informatiques ou une connaissance de la plate-forme et du méta-modèle sous-jacents pour être accessible également aux experts du domaine. En effet, dans nos expérimentations, des étudiants de maîtrise ont écrit des spécifications en moins de 15 minutes, selon leur connaissance des défauts, sans aucun besoin de compréhension du méta-modèle et de la plate-forme d'implémentation sous-jacents. Nous fournissons également un répertoire de fiches de règles qui peuvent être réutilisées et disponibles sur le site [DECOR, 2006].

De plus, comme la méthode est itérative, si un concept clef est manquant, il est possible de l'ajouter au langage. La méthode est flexible ainsi que le langage. La flexibilité des fiches de règles dépend de l'expressivité du langage et des concepts-clefs disponibles. L'expressivité du langage a été testée sur un ensemble représentatif de défauts, 8 anti-patrons et 21 défauts de code.

3.1.3 Étape 3 : Génération des algorithmes

Tout comme les deux étapes précédentes, nous décrivons l'entrée, la sortie, la description et l'implémentation de l'étape de génération des algorithmes ainsi qu'une illustration avec le Spaghetti Code suivie d'une discussion.

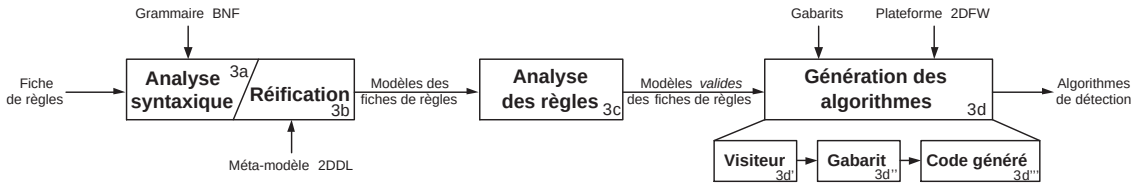
3.1.3.1 Processus

Entrée : les fiches de règles des défauts.

Sortie : les algorithmes de détection pour les défauts.

Description : le processus de génération des algorithmes se décompose en quatre étapes automatiques similaires aux étapes traditionnelles de la compilation des langages, passant des fiches de règles à leur réification, à l'analyse des règles puis à la génération des algorithmes de détection, comme le montre la figure 3.8 :

FIG. 3.8 – Processus de génération.



□

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

3a. **Analyse syntaxique.** La première étape consiste à analyser la syntaxe d'une fiche de règles. Un analyseur syntaxique construit avec JFLEX et JAVACUP[1] à partir de la grammaire BNF des fiches et augmenté avec des actions sémantiques appropriées correspondant à des instructions de code permet de construire les modèles des fiches de règles.

3b. **Réification.** Les actions sémantiques produisent, lors de l'analyse d'une fiche, un modèle de la fiche, instance du méta-modèle 2DDL (*Design Defects Definition Language*) [Moha *et al.*, 2006b]. Le méta-modèle 2DDL définit les constituants nécessaires pour représenter les fiches, règles, opérateurs ensemblistes et les propriétés. Les fiches de règles sont réifiées selon le méta-modèle 2DDL pour permettre aux algorithmes d'accéder et de manipuler de façon programmable les modèles résultants. La réification est un mécanisme important utilisé en informatique pour manipuler les concepts de façon programmable [Kiczales *et al.*, 1991].

3c. **Analyse des règles.** Des analyses de cohérence et des analyses spécifiques au domaine sont appliquées sur les modèles des fiches de règles afin de s'assurer de la validité des spécifications.

3d. **Génération des algorithmes.** Le modèle d'une fiche est enfin visité pour générer le code source associé à chaque constituant du modèle sur la base de gabarits prédéfinis. Les gabarits et le code source généré se basent sur notre plate-forme pour la détection des défauts de conception 2DFW (*Design Defects FrameWork*), qui fournit les services d'accès aux entités d'un système et à leurs propriétés.

Implémentation : la réification est automatique en utilisant l'analyseur syntaxique fourni avec le méta-modèle 2DDL. La génération est également automatique et repose sur la plate-forme 2DFW (*Design Defects FrameWork*), qui fournit des services communs aux algorithmes de détection pour des systèmes orientés objet. Ces services implémentent des opérations sur les relations, les opérateurs, les propriétés et les valeurs ordinales comme indiqué dans les fiches de règles. La plate-forme fournit également des services pour construire, accéder et analyser les modèles des systèmes. Ainsi, il est possible de calculer des métriques, analyser des relations structurelles, réaliser des analyses structurelles et lexicales et appliquer les règles. La définition des services fournis et toute la conception de la plate-forme ont été guidés par les concepts-clefs issus de l'analyse de domaine et utilisés dans le langage.

Caractéristiques principales de la génération. Notre solution au problème de la génération automatique des algorithmes de détection possède trois caractéristiques principales qui la distinguent des travaux précédents.

- La génération des algorithmes de détection est réalisée sous forme d'un visiteur sur les modèles des fiches de règles. Ce choix est similaire au choix habituel dans un compilateur de 'visiter' les noeuds de l'arbre de syntaxe abstraite pour produire du code. Ainsi, l'unique source de données pour la génération est la fiche de

[1] Plus d'information sur JFlex et JavaCUP à `http://www2.cs.tum.edu/projects/cup/`.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

règles, assurant ainsi la traçabilité des algorithmes générés et des classes candidates détectées avec la spécification du défaut.

- Afin de factoriser un maximum de code et de simplifier le visiteur, la génération utilise des gabarits de code pré-définis pour chaque type de propriétés. Ces gabarits contiennent des balises qui sont remplacées par le code spécifique à une fiche de règles lors de la génération.
- Le code source des gabarits utilise les services offerts par la plate-forme 2DFW. Ces services permettent d'accéder aux entités d'un système (dans lequel détecter les défauts) et à leurs propriétés.

Nous décrivons en détail dans la suite chacune des étapes du processus de génération donné à la figure 3.8.

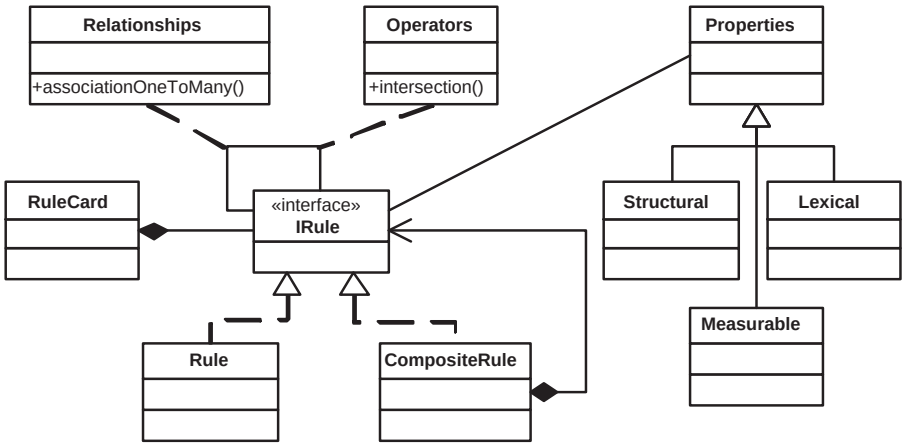
3a. Analyse syntaxique / 3b. Réification. La grammaire BNF des défauts, présentée dans la figure 3.5, est utilisée pour fournir un méta-modèle, nommé 2DDL (*Design Defects Definition Language*), et un analyseur syntaxique pour modéliser les fiches de règles et manipuler ces modèles 2DDL de façon programmable. Le méta-modèle 2DDL a été conçu pour réifier les spécifications des défauts à partir des fiches de règles. La figure 3.9 est un extrait du méta-modèle 2DDL, qui définit les constituants pour représenter les fiches de règles, les règles, les opérateurs ensemblistes, les relations entre les règles et les propriétés. Une fiche de règle est spécifiée concrètement sous forme d'une instance de la classe `RuleCard`. Une instance de `RuleCard` est composée d'objets du type `IRule`, qui décrit les règles qui peuvent être soit simples soit composites. Une règle composite, `CompositeRule`, est une règle composée d'autres règles (le patron de conception Composite [Gamma *et al.*, 1994] est utilisé à cet effet dans la conception du méta-modèle 2DDL). Les règles sont combinées en utilisant des opérateurs ensemblistes définis dans la classe `Operators`. Les relations structurelles sont étudiées en utilisant des méthodes définies dans la classe `Relationships`. Le méta-modèle 2DDL implémente également le patron de conception Visiteur. Un analyseur syntaxique étudie les fiches de règles et produit une instance de la classe `RuleCard` à partir du méta-modèle 2DDL.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.9 – Méta-modèle 2DDL.



□

3c. Analyse des règles. Cette étape consiste à visiter les modèles des fiches de règles et à appliquer des analyses de cohérence et des analyses spécifiques au domaine avant de générer les algorithmes de détection. Les analyses de cohérence consistent à vérifier que les fiches de règles ne sont pas incohérentes, redondantes ou incomplètes. Une règle incohérente correspond, par exemple, à deux défauts de code définis avec des noms identiques mais qui ont des propriétés différentes. Une règle redondante correspond, par exemple, à deux défauts de code définis avec des noms différents mais qui ont des propriétés identiques. Un exemple d'une règle incomplète correspond à un défaut de code qui est référencé dans la règle principale d'un défaut de conception mais qui n'est pas défini dans l'ensemble des règles correspondant aux défauts de code. Les analyses spécifiques au domaine consistent à vérifier que les règles sont conformes au domaine. Par exemple, la valeur associée à une métrique a une signification dans le domaine : typiquement, une propriété mesurable avec la métrique du nombre de méthodes déclarées `NMD` égale à un flottant n'a aucun sens dans le domaine.

3d. Génération des algorithmes. La génération des algorithmes de détection s'appuie sur un visiteur, des gabarits et les services fournis par la plate-forme 2DFW.

3*d*′**. Visiteur.** La génération des algorithmes de détection est implémentée sous la forme d'un visiteur sur les modèles des fiches de règles pour générer le code source approprié [Moha *et al.*, 2008a ; Moha *et al.*, 2008b].

3*d*″**. Gabarits.** Les gabarits sont des extraits de code source JAVA avec des balises bien définies à remplacer par du code concret. Pendant la visite du modèle d'une fiche comme indiqué dans la figure 3.8, nous remplaçons les balises des gabarits avec les

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

données et les valeurs appropriées provenant des fiches. Nous utilisons des gabarits car nos travaux précédents [Moha *et al.*, 2006a ; Moha *et al.*, 2006b ; Moha *et al.*, 2008a] ont montré que les algorithmes de détection ont des structures récurrentes. Ainsi, nous regroupons naturellement toutes les structures communes des algorithmes de détection au sein de gabarits, par opposition aux approches existantes, où les algorithmes sont implantés manuellement et avec peu de réutilisation. Le code source généré pour une fiche est l'algorithme de détection du défaut de conception correspondant et ce code est directement exécutable sans aucune intervention manuelle.

3*d*‴**. Code généré.** Les algorithmes de détection générés sont par construction déterministes. Nous n'avons pas besoin de réviser manuellement le code car le processus de génération assure la concordance entre le code source généré et les fiches des règles données. Le code généré tend parfois lui-même vers un Spaghetti Code car il est généré automatiquement. Cependant, il n'est pas destiné à être lu par les développeurs. Nous comptons l'améliorer et l'optimiser dans les travaux futurs. Les algorithmes de détection générés sont en JAVA. Par contre, il est possible d'appliquer ces algorithmes sur des programmes avec des langages autres que JAVA pour autant qu'il existe un analyseur syntaxique pour ces langages.

La plate-forme 2DFW. La plate-forme 2DFW est construite au-dessus du méta-modèle PADL (*Pattern and Abstract-level Description Language*) [Guéhéneuc et Antoniol, 2007] et du répertoire POM (*Primitives, Operators, Metrics*) pour calculer les métriques [Guéhéneuc *et al.*, 2004]. PADL est un méta-modèle indépendant du langage pour représenter les systèmes orientés objet [Albin-Amiot *et al.*, 2002], incluant les relations binaires de classe [Guéhéneuc et Albin-Amiot, 2004] et les accesseurs. PADL offre un ensemble d'entités (classes, interfaces, méthodes, attributs, relations, etc.) à partir desquelles des modèles de systèmes peuvent être construits. Il offre également des méthodes pour manipuler facilement ces modèles et générer d'autres modèles, en utilisant le patron de conception Visiteur. Nous avons choisi PADL car il compte six années d'actif développement et est maintenu en interne, mais nous aurions pu utiliser un autre méta-modèle ou extracteur de modèle de source code tel que cela est présenté dans [Murphy et Notkin, 1996].

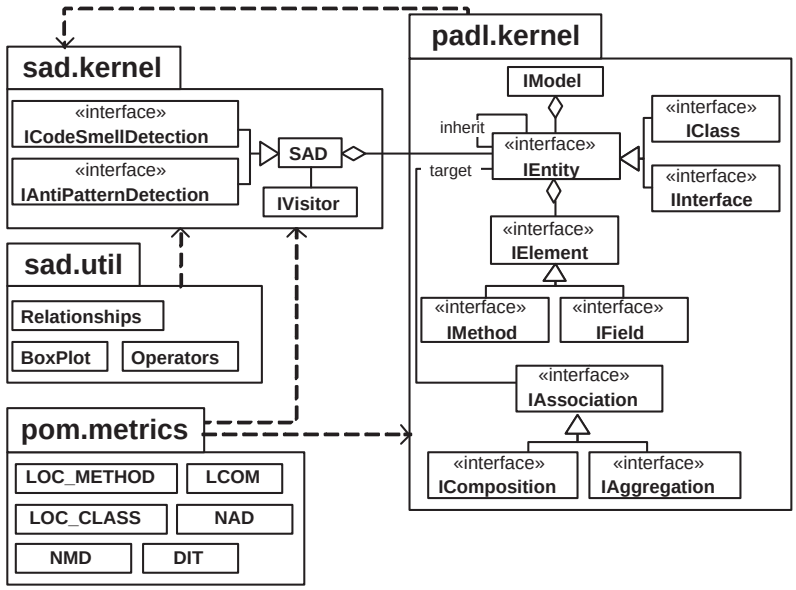
La figure 3.10 donne un aperçu de l'architecture de la plate-forme 2DFW. Cette architecture est composée de deux paquetages principaux, `sad.kernel` et `sad.util`. Le paquetage `sad.kernel` contient des classes et des interfaces noyaux. La classe `SAD` représente les défauts et est spécialisée en deux sous-classes, `AntiPattern` et `CodeSmell`. Cette hiérarchie est cohérente avec notre taxonomie des défauts. Un défaut agrège des entités définies via l'interface `IEntity` de `padl.kernel`. Par exemple, un défaut est un ensemble de classes avec des caractéristiques particulières. Les interfaces `IAntiPatternDetection` et `ICodeSmellDetection` définissent les services que les algorithmes de détection doivent fournir. Le paquetage `sad.util` déclare des classes utilitaires qui permettent de manipuler les concepts-clefs des fiches de règles.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 3.10 – Sous-ensemble de l'architecture de la plate-forme 2DFW.



□

Nous illustrons dans la suite les services offerts par la plate-forme 2DFW en utilisant des extraits de code simplifiés.

Opérateurs ensemblistes. La plate-forme 2DFW fournit la classe `Operators` dans le paquetage `sad.util` qui définit les méthodes nécessaires pour réaliser l'intersection, l'union, la différence, l'inclusion et la négation entre les défauts de code. Ces opérateurs fonctionnent sur les ensembles de classes qui sont potentiellement des défauts de code et retournent des nouveaux ensembles contenant seulement les classes appropriées.

```
1   final Set setOfLongMethodsWithNoParameter  =  CodeSmellOperators.intersection(
2                                                     setOfLongMethods,
3                                                     setOfMethodsWithNoParameter );
```

Propriétés mesurables. Les propriétés basées sur des métriques sont calculées en utilisant POM, qui fournit 44 métriques telles que le nombre de lignes de code dans une classe `LOC_CLASS`, le nombre de méthodes déclarées `NMD` ou le manque de cohésion entre les méthodes `LCOM`. POM est facilement extensible. La plate-forme 2DFW fournit, en utilisant POM dans le paquetage `pom.metrics`, un moyen pratique pour calculer toute métrique spécifiée sur un ensemble de classes.

```
1   final IClass aClass     =  iteratorOnClasses.next();
2   final double aClassLOC  =  Metrics.compute("LOC_CLASS", aClass);
```

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

La classe `BoxPlot` dans le paquetage `sad.util` calcule les quartiles pour un ensemble de valeurs d'une métrique et offre des méthodes pour y accéder et déterminer les valeurs extrêmes (en anglais, *outliers*).

```
1   final BoxPlot boxPlot   =  new BoxPlot(LOCofSetOfClasses, fuzziness);
2   final Map setOfOutliers =  boxPlot.getHighOutliers();
```

Propriétés lexicales. La vérification des propriétés lexicales est possible via l'utilisation du méta-modèle PADL, qui permet de vérifier les noms des classes, des méthodes et des attributs par rapport aux noms définis dans les fiches de règles.

```
1   String[] CTRL_NAMES     =  new String[] { "Calculate", "Display", ..., "Make" };
2
3   final IClass aClass     =  iteratorOnClasses.next();
4   for (int i = 0; i < CTRL_NAMES.length; i++) {
5                     if (aClass.getName().contains(CTRL_NAMES[i])) { ... }
6   }
```

Propriétés structurelles. Toute propriété structurelle peut être spécifiée en utilisant PADL, lequel fournit tous les constituants et les méthodes sur ces constituants pour évaluer les propriétés structurelles.

```
1   final IClass aClass     =  iteratorOnClasses.next();
2   Boolean isClassAbstract =  aClass.isAbstract();
```

Relations structurelles. Le méta-modèle PADL fournit également les constituants décrivant les relations binaires entre classes. L'existence de certaines relations entre classes peut être considérée comme un défaut, comme une association entre une classe principale et ses classes de données.

```
1   final Set setOfCandidateBlobs =  Relations.associationOneToMany(
2                                                   setOfMainClasses,
3                                                   setOfDataClasses );
```

3.1.3.2 Exemple illustratif

La figure 3.11 présente un exemple de génération d'une propriété mesurable et d'un opérateur ensembliste pour illustrer l'utilisation du visiteur, des gabarits et des services de la plate-forme 2DFW. Nous détaillons maintenant chaque caractéristique en nous appuyant sur les exemples donnés sur la figure. Nous ne présentons pas d'exemple de génération des propriétés lexicales, structurelles et relationnelles car elles sont similaires à celles illustrées.

Propriétés mesurables. La figure 3.11(a) présente la règle définie pour le défaut de code *`LongMethod`* correspondant à une propriété mesurable avec une valeur ordinale associée à la métrique `LOC_METHOD`.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Le gabarit donné sur la figure 3.11(e) est une classe appelée `<CODESMELL>Detection` qui étend la classe `CodeSmellDetection` et implémente `ICodeSmellDetection`. Cette classe déclare la méthode `performDetection()`, qui consiste à calculer la métrique spécifiée sur chaque classe du système. Toutes les valeurs des métriques sont comparées entre elles avec le boxplot (*cf.* classe `BoxPlot`) pour identifier les valeurs ordinales, *i.e.*, des valeurs extrêmes ou des valeurs normales. Ensuite, le boxplot retourne seulement les classes avec des valeurs de métriques qui vérifient les valeurs ordinales.

La figure 3.11(c) présente le processus de génération du code pour vérifier une propriété mesurable définie dans la fiche de règles du Spaghetti Code sur un ensemble de constituants. Lorsque la fiche est visitée sur le modèle de la fiche de règles, nous remplaçons la balise
`<CODESMELL>` par le nom de la règle *`LongMethod`*, la balise `<METRIC>` par le nom de la métrique `LOC_METHOD`, la balise `<FUZZINESS>` par la valeur 10.0 correspondant au degré de logique floue et la balise `<ORDINAL_VALUES>` par la méthode associée avec la valeur ordinale `VERY_HIGH`.

La figure 3.11(g) présente le code source généré correspondant au résultat de la génération à partir de la règle donnée sur la figure 3.11(a).

Opérateurs. La figure 3.11(b) présente une règle combinant deux autres règles en utilisant l'opérateur d'intersection. La génération du code pour les opérateurs entre règles dans une fiche est légèrement différente de la génération pour les propriétés : le gabarit présenté dans la figure 3.11(f) contient également une classe appelée `<CODESMELL>Detection` qui étend la classe `CodeSmellDetection` et implémente `ICodeSmellDetection`. Mais, la méthode `performDetection()` consiste à combiner avec un opérateur ensembliste la liste des classes de chaque opérande de la règle donnée à la figure 3.11(b) et à retourner les classes qui satisfont cette combinaison. La figure 3.11(d) présente le processus relié à la génération de code pour les opérateurs ensemblistes dans la fiche du Spaghetti Code. Quand un opérateur est visité dans le modèle de la fiche, les balises associées aux opérandes de la fiche, `operand1: LongMethod, operand2: NoParameter` et la balise `<OPERATION>` sont remplacées par le type d'opérateur ensembliste spécifié dans la fiche, *i.e.*, l'intersection. Les opérandes correspondent aux algorithmes de détection générés (*cf.* Figure 3.11(h)).

3.1.3.3 Discussions

L'implantation d'un générateur d'algorithmes n'est pas en soi un réel défi car la génération de code est un problème connu [Fraser *et al.*, 1992]. Par contre, le problème que nous résolvons est le passage automatique des spécifications des défauts aux algorithmes de détection. Ainsi, nous évitons une construction manuelle des algorithmes, qui est coûteuse et peu réutilisable, et nous assurons la traçabilité entre les spécifications et les occurrences des défauts détectés.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

(a) Extrait du Spaghetti Code.

```
RULE:LongMethod {METRIC LOC_METHOD
    VERY_HIGH 10.0};
```

(b) Extrait du Spaghetti Code.

```
RULE:Inter1
    {INTER LongMethod NoParameter};
```

(c) Visiteur.

```
public void visit(IMetric aMetric)} {
  replaceTAG("<CODESMELL>", aRule.getName());
  replaceTAG("<METRIC>", aMetric.getName());
  replaceTAG(<FUZZINESS>,
    aMetric.getFuzziness());
  replaceTAG(<ORDINAL_VALUE>,
    aMetric.getOrdinalValue());
}
private String getOrdinalValue(int value) {
  switch (value) {
    case VERY_HIGH : "getHighOutliers";
    case HIGH      : "getHighValues";
    case MEDIUM    : "getNormalValues";
    ...
  }
```

(d) Visiteur.

```
public void visit(IOperator anOperator) {
  replaceTAG("<OPERAND1>",
    anOperator.getOperand1());
  replaceTAG("<OPERAND2>",
    anOperator.getOperand2());
  switch (anOperator.getOperatorType()) {
    case OPERATOR_UNION :
      operator = "union";
    case OPERATOR_INTER :
      operator = "intersection";
    ...
  }
  replaceTAG("<OPERATION>", operator);
```

(e) Gabarit.

```
public class <CODESMELL>Detection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public Set performDetection() {
    IClass c = iteratorOnClasses.next();
    LOCofSetOfClasses.add(
      Metrics.compute(<METRIC>, c));
    ...
    BoxPlot boxPlot = new BoxPlot(
      <METRIC>ofSetOfClasses, <FUZZINESS>);
    Map setOfOutliers =
      boxPlot.<ORDINAL_VALUE>();
    ...
    suspiciousCodeSmells.add( new CodeSmell(
      <CODESMELL>, setOfOutliers));
    ...
    return suspiciousCodeSmells;
  }
```

(f) Gabarit.

```
public class <CODESMELL>Detection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public void performDetection() {
    ICodeSmellDetection cs<OPERAND1> =
      new <OPERAND1>Detection();
    op1.performDetection();
    Set set<OPERAND1> =
      cs<OPERAND1>.listOfCodeSmells();
    ICodeSmellDetection cs<OPERAND2> =
      new <OPERAND2>Detection();
    op2.performDetection();
    Set set<OPERAND2> =
      cs<OPERAND2>.listOfCodeSmells();
    Set setOperation = Operators.getInstance().
      <OPERATION>(set<OPERAND1>, set<OPERAND2>);
    this.setSetOfSmells(setOperation);
  }
```

(g) Code généré.

```
public class LongMethodDetection
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public Set performDetection() {
    IClass c = iteratorOnClasses.next();
    LOCofSetOfClasses.add(
    Metrics.compute("LOC_METHOD", c));
    ...
    BoxPlot boxPlot = new BoxPlot(
      LOC_METHODofSetOfClasses, 10.0);
    Map setOfOutliers =
      boxPlot.getHighOutliers();
    ...
    suspiciousCodeSmells.add( new CodeSmell(
      LongMethod, setOfOutliers));
    ...
    return suspiciousCodeSmells;
  }
```

(h) Code généré.

```
public class Inter1
  extends CodeSmellDetection
  implements ICodeSmellDetection {
  public void performDetection() {
    ICodeSmellDetection csLongMethod =
      new LongMethodDetection();
    csLongMethod.performDetection();
    Set setLongMethod =
      csLongMethod.listOfCodeSmells();
    ICodeSmellDetection csNoParameter =
      new NoParameterDetection();
    csNoParameter.performDetection();
    Set setNoParameter =
      csNoParameter.listOfCodeSmells();
    Set setOperation = Operators.getInstance().
      intersection(setLongMethod, setNoParameter);
    this.setSetOfSmells(setOperation);
  }
```

FIG. 3.11 – Génération pour les propriétés mesurables (à gauche) et les opérateurs ensemblistes (à droite).

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Le méta-modèle 2DDL et la plate-forme 2DFW, à l'aide du méta-modèle PADL et de la plate-forme POM, fournissent les mécanismes concrets pour générer et appliquer les algorithmes de détection. Cependant, en appliquant DECOR, il est possible de concevoir un autre langage et de construire un autre méta-modèle pour modéliser les fiches de règles. Les algorithmes de détection peuvent être générés via d'autres plates-formes telles que les outils présentés dans l'état de l'art.

L'ajout d'une nouvelle propriété dans le DSL nécessite d'implémenter explicitement l'analyse correspondante au sein de la plate-forme 2DFW. Nous avons expérimenté l'ajout de nouvelles propriétés et un nouvel ajout nécessitait entre 15 minutes à une journée par développeur selon la difficulté de l'analyse. Par contre, cette opération n'est nécessaire qu'une fois par propriété manquante et est très rare car nous fournissons un langage suffisamment riche pour décrire un grand nombre de défauts.

La création des modèles 2DDL est répétitive pour chaque défaut mais le méta-modèle 2DDL et la plate-forme 2DFW sont génériques et n'ont pas besoin d'être redéfinis. Les modèles des systèmes sont construits avant d'appliquer l'algorithme de détection, alors que les valeurs des métriques sont calculées à la volée et à la demande.

3.1.4 Étape 4 : Détection

Pour décrire cette dernière étape de notre technique DETEX, nous utilisons le même patron que les trois étapes précédentes incluant l'entrée, la sortie, la description et l'implémentation de l'étape ainsi qu'une illustration avec le Spaghetti Code suivie d'une discussion.

3.1.4.1 Processus

Entrée : les algorithmes de détection pour les défauts et le modèle d'un système où détecter les défauts.

Sortie : les classes suspectes dont les propriétés et les relations sont conformes aux modèles des défauts utilisés pour générer les algorithmes de détection.

Description : les algorithmes de détection sont appliqués automatiquement sur un ou plusieurs modèles de systèmes soit de manière isolée soit en lot sur plusieurs modèles. Plus concrètement, la détection correspond à une recherche de forme définie par les algorithmes de détection sur les modèles qui représentent les entités des systèmes analysés. Les algorithmes de détection générés sont utilisés pour détecter des classes suspectes.

Implémentation : l'appel aux algorithmes de détection générés est direct en utilisant les services fournis par la plate-forme 2DFW. Le modèle d'un système est obtenu en se

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

basant sur les constituants du méta-modèle PADL [Albin-Amiot *et al.*, 2002]. Un sous-ensemble de ce méta-modèle est représenté dans le paquetage `padl.kernel` de la figure 3.10. Ce modèle est obtenu par rétro-ingénierie.

3.1.4.2 Exemple illustratif

Dans le programme JAVA libre XERCES v2.7.0, 76 classes suspectes Spaghetti Code ont été trouvées parmi les 513 classes du système.

Le code nécessaire pour réaliser la détection du Spaghetti Code est synthétisé ci-dessous. Ce code consiste à créer une instance de type `SpaghettiCodeDetection` sur le modèle du système à analyser. La classe `SpaghettiCodeDetection` correspond à l'algorithme de détection associé au Spaghetti Code. Ensuite, la détection est exécutée via la méthode `performDetection()`, puis la liste des anti-patrons détectés de type Spaghetti Code est affichée :

```
1  IAntiPatternDetection antiPatternDetection =
2    new SpaghettiCodeDetection(model);
3  antiPatternDetection.performDetection();
4  ...
5  outputFile.println(
6    antiPatternDetection.getSetOfAntiPatterns());
```

3.1.4.3 Discussion

Les modèles sur lesquels les algorithmes de détection ont été appliqués peuvent être obtenus via une ingénierie directe ou une rétro-ingénierie du code source car les conceptions industrielles sont rarement disponibles librement. En plus, les documents de conception, tout comme la documentation en général, sont souvent obsolètes. Ainsi, dans de nombreux systèmes avec une documentation insuffisante, le code source est la seule source d'information fiable [Muller *et al.*, 2000] ainsi que la source de données la plus précise et la plus à jour. C'est pourquoi, parce que l'efficacité de la détection dépend des données disponibles sur un système, nous avons choisi de travailler avec des données obtenues par rétro-ingénierie car cela est plus commode et fournit un accès à plus de données que les seuls diagrammes de classes, en particulier les invocations de méthodes. DETEX pourrait aussi s'appliquer aux diagrammes de classes, bien que certaines règles ne seraient plus valides. Ainsi, nous n'avons pas analysé directement les diagrammes de classes bien que possible et considéré comme travaux futurs.

3.2 Expérimentations

Nous validons DETEX et indirectement DECOR en étudiant à la fois l'application de ces quatre étapes et les résultats de la détection sur 4 défauts de conception et leurs 15 défauts de code associés sur 11 systèmes libres. La validation est réalisée par des

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

ingénieurs logiciels indépendants, qui doivent évaluer si les classes suspectes sont des défauts, selon les contextes des systèmes.

3.2.1 Hypothèses des expérimentations

Nous voulons valider les trois hypothèses suivantes qui vérifient l'efficacité de DETEX et indirectement l'utilité de DECOR.

1. *Le* DSL *permet de spécifier plusieurs défauts différents.* Cette hypothèse vérifie la pertinence des spécifications et la validité d'application de notre méthode sur quatre défauts de conception, composé de 15 défauts de code.
2. *Les algorithmes de détection générés ont un rappel de 100%, c'est-à-dire tous les défauts de conception sont détectés, et une précision supérieure à 50%, c'est-à-dire les algorithmes de détection reportent moins de la moitié de faux positifs par rapport au nombre de vrais positifs.* Étant donné le compromis à faire entre la précision et le rappel, nous supposons qu'une précision de 50% est assez significatif par rapport à un rappel de 100%. L'hypothèse vérifie la précision des fiches de règles et la pertinence de la génération des algorithmes. Elle vérifie aussi la pertinence des services fournis par la plate-forme 2DFW.
3. *La complexité des algorithmes générés est raisonnable, c'est-à-dire les temps de calculs sont de l'ordre de la minute.* L'hypothèse vérifie la complexité des algorithmes de détection.

3.2.2 Sujets des expérimentations

Nous utilisons notre méthode pour décrire quatre anti-patrons bien connus mais différents du livre de Brown *et al.* [1998] : Blob [page 73], Functional Decomposition [page 97], Spaghetti Code [page 119] et Swiss Army Knife [page 197]. Le tableau 3.1 page 38 résume chacun de ces anti-patrons. Ces défauts de conception incluent dans leurs spécifications 15 défauts de code différents, certains sont décrits dans le livre de Fowler [1999]. Nous générons automatiquement les algorithmes de détection associés.

3.2.3 Processus des expérimentations

Nous validons les résultats des algorithmes de détection en analysant les classes suspectes dans le contexte du modèle complet du système. Cette validation consiste à identifier les classes suspectes considérées comme de vraies positives par rapport au contexte du système et les fausses négatives, c'est-à-dire les classes ayant des défauts mais qui ne sont pas reportées par les algorithmes.

Ainsi, nous projetons la validation dans le domaine de l'extraction d'information et utilisons les mesures de précision et de rappel, où la précision évalue le nombre de défauts réels parmi les défauts détectés et le rappel évalue le nombre de défauts détectés

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

parmi les défauts réels dans le système, selon les définitions suivantes [Frakes et Baeza-Yates, 1992] :

$$\text{précision} = \frac{|\{\text{défauts existants}\} \cap \{\text{défauts détectés}\}|}{|\{\text{défauts détectés}\}|}$$

$$\text{rappel} = \frac{|\{\text{défauts existants}\} \cap \{\text{défauts détectés}\}|}{|\{\text{défauts existants}\}|}$$

Nous avons sollicité l'aide d'ingénieurs logiciels pour calculer de façon indépendante le rappel des algorithmes générés. La validation des défauts détectés se fait manuellement car seuls les ingénieurs logiciels peuvent évaluer si une classe suspecte est *réellement* un défaut ou une fausse positive en se basant sur la description du défaut dans la littérature, le contexte et les caractéristiques du système. Cette étape peut prendre beaucoup de temps si les spécifications des défauts ne sont pas assez contraignantes car le nombre de classes suspectes peut être important.

3.2.4 Objets des expérimentations

Les objets des expérimentations sont des modèles obtenus par rétro-ingénierie de 10 systèmes JAVA libres : ARGOUML, AZUREUS, GANTTPROJECT, LOG4J, LUCENE, NUTCH, PMD, QUICKUML et deux versions de XERCES. Nous utilisons des systèmes librement disponibles pour faciliter les comparaisons et les réplications de nos expérimentations. Nous fournissons des informations sur ces systèmes dans le tableau 3.2. Nous avons aussi appliqué les algorithmes sur ECLIPSE et discutons les résultats.

3.2.5 Résultats des expérimentations

Nous reportons les données expérimentales en trois étapes. Tout d'abord, nous reportons les précisions et les rappels des algorithmes de détection sur XERCES v2.7.0 pour les quatre défauts de conception (Blob, Functional Decomposition, Spaghetti Code, Swiss Army Knife). Ces données constituent le premier rapport sur la précision et le rappel reportés via une technique de détection. Ensuite, nous reportons les détections de ces défauts de conception et leurs précisions sur dix systèmes libres par rétro-ingénierie et nous illustrons par des exemples concrets les résultats de ces détections. Enfin, nous avons également appliqué nos algorithmes de détection sur ECLIPSE v3.1.2, démontrant son passage à l'échelle, c'est-à-dire l'aptitude à s'exécuter correctement sur un très grand ensemble de données, et mettant en valeur le problème de balance entre les nombres de classes suspectes et les précisions.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Tableau 3.2 – Liste des systèmes.

Nom	Version	Lignes de code	Nombre de classes	Nombre d'interfaces
ARGOUML	0.19.8	113 017	1 230	67
Un outil de modélisation UML				
AZUREUS	2.3.0.6	191 963	1 449	546
Un client pair-à-pair implémentant le protocol BitTorrent				
GANTTPROJECT	1.10.2	21 267	188	41
Un outil de gestion de projet pour réaliser des diagrammes de GANTT				
LOG4J	1.2.1	10 224	189	14
Un framework extensible pour logger et débugger les applications Java				
LUCENE	1.4	10 614	154	14
Un moteur de recherche textuelle en JAVA				
NUTCH	0.7.1	19 123	207	40
Un moteur de recherche Web basé sur LUCENE				
PMD	1.8	41 554	423	23
Un outil d'analyse de code source et détection de bugs				
QUICKUML	2001	9 210	142	13
Un outil de modélisation UML pour les diagrammes de classes et de séquences				
XERCES	1.0.1	27 903	189	107
Un plate-forme pour construire des analyseurs syntaxiques XML en JAVA				
XERCES	2.7.0	71 217	513	162
Version 2.7.0 de l'analyseur syntaxique XERCES				

□

3.2.5.1 Résultats sur XERCES pour les précisions et les rappels

Nous avons demandé à trois étudiants de maîtrise [2] et deux développeurs indépendants d'analyser manuellement XERCES en utilisant seulement les livres de Brown et Fowler et leur propre compréhension pour identifier les défauts de conception et de calculer la précision et le rappel sur les classes suspectes. Ils ont utilisé un environnement de développement intégré tel que ECLIPSE pour visualiser le code source et étudier chaque classe l'une après l'autre. Chaque fois qu'il y avait un doute sur une classe candidate, ils ont pris les livres comme référence pour décider par consensus si oui ou non la classe était réellement un défaut de conception. Ils ont réalisé une étude minutieuse de XERCES et produit un fichier XML contenant les classes suspectes pour les quatre défauts de conception. Cette tâche étant fastidieuse, certains défauts de conception peuvent avoir été manqués par erreur, c'est pourquoi il peut s'avérer nécessaire de réaliser à nouveau cette même tâche avec d'autres développeurs sur XERCES pour confirmer les résultats et sur d'autres systèmes pour augmenter la base de données des défauts.

[2] La maîtrise au Canada est équivalent à un master 2 recherche en France.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Tableau 3.3 – Précision et rappel dans XERCES v2.7.0. (*entre parenthèses, le pourcentage des classes affectées par un défaut de conception,* XERCES *v2.7.0 contient 513 classes*).

Défauts	Nombres de vrais positifs		Nombres de défauts détectés		Précision	Rappel	Temps de détection
Blob	39/513	(7.6%)	44/513	(8.6%)	88.6%	100.00%	2.45s
Functional Decomposition	15/513	(3.0%)	29/513	(5.6%)	51.7%	100.00%	0.16s
Spaghetti Code	46/513	(9.0%)	76/513	(15%)	60.5%	100.00%	0.22s
Swiss Army Knife	23/513	(4.5%)	56/513	(11%)	41.1%	100.00%	0.05s
					60.5%	100%	0.72s

□

Le tableau 3.3 reporte la précision et le rappel de la détection des quatre défauts de conception dans XERCES v2.7.0. Nous avons réalisé tous les calculs sur un Intel Dual Core à 1.67GHz avec 1Gb de RAM. Les temps de calcul n'incluent pas la construction des modèles du système, qui est de 31 secondes en moyenne, mais incluent les accès pour calculer les métriques et pour vérifier les relations structurelles et les propriétés lexicales et structurelles. Les temps de détection peuvent varier d'un défaut à un autre selon la complexité des règles spécifiées pour détecter le défaut. Par exemple, les règles utilisées pour la détection du Swiss Army Knife sont bien plus simples que celles utilisées dans le Blob, d'où le petit temps de détection.

Les rappels de nos algorithmes de détection sont de 100% pour chaque défaut de conception. Nous avons spécifié les règles de détection de telle sorte à avoir un rappel parfait pour voir son impact sur la précision. Dans notre validation expérimentale, le rappel représente la variable indépendante alors que la précision est la variable dépendante. Ainsi, les résultats obtenus fourniront une bonne base de comparaison pour les autres approches de détection. Les précisions se situent entre 41.1% et près de 90% (avec une précision totale de 60.5%), représentant entre 5.6% et 15% du nombre total de classes, ce qu'il est raisonnable pour un développeur d'analyser à la main, par rapport à analyser le système en entier—513 classes—manuellement. Le nombre de classes suspectes obtenues est donc généralement d'un ordre de magnitude plus petit que le nombre total de classes dans le système ; ainsi, notre méthode facilite en effet l'inspection de code par les ingénieurs logiciels.

3.2.5.2 Exemple illustratif

Pour le Spaghetti Code, nous avons trouvé 76 classes suspectes. Sur les 76 classes suspectes, 46 sont en fait des Spaghetti Code précédemment identifiés dans XERCES manuellement par les développeurs indépendants, ce qui a mené à une précision de 60.5% et un rappel de 100% (*cf.* la troisième ligne dans le tableau 3.3).

Le fichier résultat contient toutes les classes suspectes, incluant la classe `org.apache.xerces.xinclude.XIncludeHandler` déclarant 112 méthodes. Parmi ces 112 métho-

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

des, la méthode `handleIncludeElement` est typique d'un Spaghetti Code car cette méthode n'utilise pas l'héritage et le polymorphisme, déclare 18 variables globales et compte 759 LOC, alors que la longueur maximale moyenne d'une méthode dans le système en utilisant le boxplot est de 254.5 LOC. Un autre exemple de Spaghetti Code est la classe `org.apache.xerces.impl.xpath.regex.RegularExpression` déclarant la méthode `matchCharArray(Context,Op,int,int,int)` qui a une taille de 1 246 LOC.

Le fichier résultat, qui contient toutes les classes suspectes, se présente comme suit :

```
1.100.Name = SpaghettiCode                  2.100.Name = SpaghettiCode
1.100.Class = org.apache.xerces.xinclude.   2.100.Class = org.apache.xerces.impl.
  XIncludeHandler                             xpath.regex.RegularExpression
1.100.NoInheritance.DIT-0 = 1.0             2.100.NoInheritance.DIT-0 = 1.0
1.100.LongMethod.MethodName =               2.100.LongMethod.MethodName =
  handleIncludeElement(XMLAttributes)         matchCharArray(Context,Op,int,int,int)
1.100.LongMethod.LOC_METHOD = 759.0         2.100.LongMethod.LOC_METHOD = 1246.0
1.100.LongMethod.LOC_METHOD_Max = 254.5     2.100.LongMethod.LOC_METHOD_Max = 254.5
1.100.GlobalVariable-0 = SYMBOL_TABLE       2.100.GlobalVariable-0 = WT_OTHER
1.100.GlobalVariable-1 = ERROR_REPORTER     2.100.GlobalVariable-1 = WT_IGNORE
1.100.GlobalVariable-2 = ENTITY_RESOLVER    2.100.GlobalVariable-2 = EXTENDED_COMMENT
1.100.GlobalVariable-3 = BUFFER_SIZE        2.100.GlobalVariable-3 = CARRIAGE_RETURN
1.100.GlobalVariable-4 = PARSER_SETTINGS    2.100.GlobalVariable-4 = IGNORE_CASE
...                                         ...
```

Les résultats de la détection reportés dans ce fichier peuvent également être visualisés de manière interactive via l'interface de PTIDEJ [Guéhéneuc, 2005], outil dans lequel la technique de détection DETEX a été intégrée. La visualisation des défauts détectés dans PTIDEJ consiste à mettre en valeur dans un pseudo-diagramme de classes UML toutes les classes du système analysé qui participent à l'occurrence d'un défaut. Dans le cas du Spaghetti Code, seule une seule classe est affectée par ce défaut. Par contre, dans le cas du Blob, plusieurs classes dont la large classe et les classes de données contribuent à l'occurrence de ce défaut. Ces différentes classes sont donc toutes listées dans le fichier résultat et mises en valeur au niveau du diagramme de classes dans PTIDEJ.

Si nous passons en revue le code source, nous observons que la méthode `matchCharArray` contient un opérateur de comparaison et du code dupliqué pour 20 opérateurs différents (tels que =, <, >, [a-z], etc.) dans des expressions régulières et la classe `org.apache.xerces.impl.xpath.regex.Op` a actuellement des sous-classes pour la plupart de ces opérateurs. La méthode `matchCharArray` pourrait avoir été implémentée dans un style plus orienté objet en distribuant l'opérateur de comparaison dans les sous-classes `Op`. Ceci distribue la longue méthode en plusieurs méthodes plus petites, qui seront réparties dans toutes les sous-classes `Op`. Cependant, une telle conception introduirait, dans l'appel principal de la méthode, des appels polymorphiques en traversant tous les caractères du tableau correspondant aux différents opérateurs. Une raison valide des concepteurs de XERCES pour ne pas avoir opté pour une telle "meilleure" conception est de favoriser la performance, probablement au coût de la maintenabilité.

Les 46 Spaghetti Code représentent des vrais positifs et incluent des "mauvais" Spaghetti Code tel que la méthode `handleIncludeElement` mais aussi de "bons" Spaghetti Code tel que la méthode `matchCharArray`. Les "bons" défauts ne sont pas rejetés car ils peuvent représenter des points chauds en terme de qualité et de maintenance.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

D'autres exemples typiques de Spaghetti Code détectés et évalués comme de vrais positifs sont les classes qui contiennent du code généré automatiquement, par exemple des analyseurs syntaxiques créés par des générateurs. Les 30 autres classes suspectes ont été rejetées par les ingénieurs logiciels et correspondent à des fausses positives. Même si ces classes vérifiaient les caractéristiques du Spaghetti Code, la plupart étaient faciles à comprendre et donc, étaient considérées comme des fausses positives. Ainsi, il semble être nécessaire d'ajouter d'autres règles ou de modifier les règles existantes pour cibler seulement les réels Spaghetti Code, par exemple, en détectant les instructions `if` et les boucles imbriquées, caractéristiques d'un code compliqué à comprendre.

3.2.5.3 Résultats sur d'autres systèmes

Le tableau 3.4 fournit, pour les neuf autres systèmes plus XERCES v2.7.0, les nombres de classes suspectes dans la première ligne de chaque colonne ; les nombres de vrais défauts de conception dans la seconde ligne ; les précisions dans la troisième ligne ; et les temps de détection dans la quatrième ligne. Nous reportons ici seulement les précisions. Les rappels sur d'autres systèmes que XERCES font partie de travaux futurs à cause de la nécessité d'analyses manuelles, qui demandent beaucoup de temps et doivent être effectuées par des ingénieurs logiciels indépendants. Nous avons également réalisé tous les calculs sur un Intel Dual Core à 1.67GHz avec 1Gb de RAM.

3.2.5.4 Illustrations des résultats

Nous présentons brièvement des exemples des quatre défauts de conception. Dans XERCES v2.7.0, la méthode `handleIncludeElement(XMLAttributes)` de la classe `org.apache.xerces.xinclude.XIncludeHandler` est un exemple typique de Spaghetti Code. Un bon exemple de défaut de conception Blob est la classe `com.aelitis.azureus.core.dht.control.impl.DHTControlImpl` dans AZUREUS. Cette classe déclare 54 attributs et 80 méthodes pour 2 965 lignes de code. Un exemple intéressant du défaut de conception Functional Decomposition est la classe `org.argouml.uml.cognitive.critics.Init` de ARGOUML, en particulier car le nom de la classe suggère une programmation fonctionnelle. La classe `org.apache.xerces.impl.dtd.DTDGrammar` dans XERCES est un exemple frappant de Swiss Army Knife car elle implémente quatre ensembles différents de services avec 71 attributs et 93 méthodes pour 1 146 lignes de code.

3.2.5.5 Résultats sur ECLIPSE et le passage à l'échelle

Nous avons également appliqué nos algorithmes de détection sur l'environnement de développement libre ECLIPSE pour illustrer le passage à l'échelle, c'est-à-dire l'aptitude à s'exécuter correctement sur un très grand ensemble de données. ECLIPSE v3.1.2 compte 2 538 774 lignes de code pour 9 099 classes et 1 850 interfaces. Il est d'un ordre de magnitude plus grand que le plus grand des systèmes libres présentés jusqu'à présent,

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

AZUREUS. La détection des quatre défauts de conception dans ECLIPSE nécessite plus de temps et produit plus de résultats. Nous détectons 848, 608, 436 et 520 classes suspectes pour les défauts de conception Blob, Functional Decomposition, Spaghetti Code et Swiss Army Knife, respectivement. Les détections prennent environ 1h20 pour chaque défaut de conception avec environ une heure pour construire le modèle. L'utilisation des algorithmes de détection sur ECLIPSE illustre le passage à l'échelle de nos algorithmes et met en valeur le problème de balance entre les nombres de classes suspectes et les précisions. En effet, si le choix est de maximiser le rappel, le nombre de classes suspectes risque d'être important d'autant plus dans les grands systèmes tels qu'ECLIPSE, et donc la précision sera faible. Et inversement, si le choix est de minimiser le nombre de fausses classes suspectes alors la précision sera élevée mais le rappel risque d'être faible. De plus, ceci démontre l'importance de spécifier les défauts dans le contexte du système dans lequel ils sont détectés. En effet, le plus grand nombre de classes suspectes pour le Blob dans ECLIPSE, environ $1/10^{ième}$ du nombre total de classes, provient probablement des choix de conception et d'implémentation et des contraintes définis au sein de la communauté ECLIPSE et donc, les spécifications des défauts devraient être adaptées pour prendre en compte ces choix. Avec notre méthode et son implémentation de référence, les ingénieurs logiciels peuvent facilement re-spécifier les défauts pour les adapter à leur contexte et obtenir une meilleure précision.

3.2.6 Discussion des résultats

Nous vérifions chacune des trois hypothèses en utilisant les résultats de la validation pour évaluer DETEX et valider, indirectement, la méthode DECOR.

1. *Le* DSL *permet de spécifier plusieurs défauts différents.* Nous avons décrit quatre défauts de conception appartenant aux catégories inter- et intra-classes et aux catégories structurelles, lexicales et mesurables, présentées dans la figure 3.3 page 41 ; ces défauts sont caractérisés par 15 défauts de code appartenant également à 6 catégories différentes, présentées dans la figure 3.2 page 40. Ainsi, nous montrons que nous pouvons décrire différents types de défauts, ce qui soutient la généralité de notre méthode et du DSL.
2. *Les algorithmes de détection générés ont un rappel de 100% et une précision supérieure à 50%.* Le tableau 3.3 page 62 montre que la précision et le rappel pour XERCES v2.7.0 correspondent à nos hypothèses avec une précision de 60.5% et un rappel de 100%. Le tableau 3.2.5.2 page 70 présente les précisions pour les neuf autres systèmes, lesquelles respectent quasiment nos hypothèses, avec une précision supérieure à 50% (sauf pour deux systèmes), ainsi validant l'utilité de notre méthode.
3. *La complexité des algorithmes générés est raisonnable, c'est-à-dire les temps de calculs sont de l'ordre de la minute.* Les temps de calcul sont en général inférieurs à quelques secondes (sauf pour ECLIPSE) car la complexité de nos algorithmes de détection dépend seulement du nombre de classes dans le système analysé, n, et du nombre de propriétés pour vérifier chaque classe. La complexité des algorithmes de détec-

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

tion générés est de $(c + op) \times \mathcal{O}(n)$, où c est le nombre de propriétés et op le nombre d'opérateurs.

Les temps de calcul pour les défauts de conception varient avec les défauts et les systèmes. Durant la validation, nous avons noté que la construction des modèles des systèmes occupe la plupart du temps de calcul, alors que les algorithmes de détection ont des temps d'exécution assez courts, ce qui explique les différences mineures entre les systèmes, sur la même ligne dans le tableau 3.2.5.2, et les différences entre les défauts de conception, dans les différentes colonnes. Les temps de calcul pour les modèles PADL ne sont pas surprenants car les modèles contiennent un nombre important de données, incluant les relations binaires entre classes [Guéhéneuc et Albin-Amiot, 2004] et les accesseurs.

Les précisions varient également en relation avec le défaut de conception et le système, comme le montre le tableau 3.2.5.2 : tout d'abord, les systèmes ont été développés dans des contextes différents et peuvent avoir des qualités de conception différentes. Des systèmes tels que AZUREUS ou XERCES peuvent être de moindre qualité que LUCENE ou QUICKUML, menant ainsi à des nombres plus importants de classes suspectes étant de réels défauts. Cependant, le faible nombre de défauts détectés dans LUCENE et QUICKUML mène à une faible précision. Par exemple, seul un Functional Decomposition a été détecté dans LUCENE mais il s'agit d'un faux positif menant ainsi à une précision de 0% et une précision moyenne de 38.2%. Ensuite, les spécifications des défauts de conception sous forme de fiches de règles peuvent être très ou peu contraignantes. Par exemple, les fiches de règles du Blob et du Spaghetti Code spécifient les défauts de manière stricte en utilisant des métriques et des relations structurelles, menant à un petit nombre de classes suspectes et à de grandes précisions. De plus, les fiches de règles du Functional Decomposition et du Swiss Army Knife spécifient ces défauts de manière assez large en utilisant des données lexicales, menant à de plus faibles précisions. Ainsi, les spécifications ne doivent pas être trop larges, pour ne pas détecter trop de classes suspectes, ni trop contraignantes et manquer des défauts. Avec DETEX, les ingénieurs logiciels peuvent raffiner les spécifications des défauts de manière systématique selon les classes suspectes détectées et leur connaissance du système et de son implémentation. Le choix des métriques et des seuils est laissé aux experts du domaine car eux seuls peuvent spécifier les défauts en prenant en compte le contexte et les caractéristiques des systèmes analysés.

Le nombre de faux positifs semble assez important ; cependant, dans ces expérimentations, si nous obtenons autant de faux positifs c'est parce que notre objectif était d'avoir 100% de rappel sur tous les systèmes. Grâce à DETEX et son DSL, les règles peuvent être affinées et modifiées systématiquement et facilement pour s'adapter aux contextes spécifiques des systèmes analysés et ainsi augmenter la précision si désiré, probablement aux dépens du rappel. Ainsi, le nombre de faux positifs sera plus bas et les ingénieurs logiciels ne perdront pas leur temps à vérifier la quantité importante de faux résultats. Dans des travaux futurs, nous proposons de trier les résultats selon un ordre critique, c'est-à-dire selon les classes qui sont le plus suspectées d'être des défauts, afin d'aider les ingénieurs logiciels à récolter et évaluer les résultats.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

3.2.7 Menaces à la validité

Validité interne. Il n'y a pas de menace à la validité interne dans nos expérimentations autre que les spécifications des défauts qui ont mené aux résultats obtenus. De plus, nous avons utilisé pour les expérimentations un ensemble représentatif de défauts pour limiter au maximum leur influence sur les résultats.

Validité externe. Une menace à la validité externe est l'utilisation exclusive de systèmes JAVA libres. Il est possible que le processus de développement libre biaise le nombre de défauts de conception, en particulier dans le cas de systèmes matures tels que PMD v1.8 ou XERCES v2.7.0. Il est également possible que le langage de programmation JAVA affecte les choix de conception et donc la présence de défauts. Cependant, nous avons appliqué nos algorithmes sur des systèmes de tailles et de qualités variables pour exclure la possibilité pour tous les systèmes d'être soit bien ou mal implémentés. De plus, nous avons choisi de réaliser une validation sur des systèmes libres afin de permettre la comparaison et la réplication de nos expérimentations. Nous sommes également en contact avec des sociétés telles que la SNCF pour répliquer ces expérimentations sur leurs systèmes commerciaux.

Validité de construction. La nature subjective d'identifier ou de spécifier des défauts et d'évaluer les classes suspectes comme défauts est une menace à la validité de construction. En effet, notre compréhension de ce que sont les défauts peut différer de celle d'un autre ingénieur logiciel. Nous atténuons cette menace en spécifiant les défauts par rapport à la littérature générale et en tirant notre inspiration des travaux précédents ; nous avons demandé aux ingénieurs logiciels en charge de calculer le rappel et la précision de faire de même. De plus, nous avons contacté les développeurs de chacun des systèmes de ces expérimentations afin de nous aider à valider les précisions et les rappels que nous avons obtenus lors de ces expérimentations. Nous avons reçu quelques réponses mais avec un intérêt fort enthousiaste. Les ingénieurs logiciels ont analysé de façon indépendante nos résultats pour LOG4J, LUCENE, PMD et QUICKUML, et ont confirmé les résultats du tableau 3.2.5.2. Nous remercions M. Adamovic, C. Alphonce, D. Cutting, T. Copeland, P. Gardner, E. Ross et Y. Shapira pour leur précieuse aide.

Validité de conclusion. Les résultats et conclusions de cette étude semblent raisonnables, en particulier ils sont basés sur une intuition commune.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Bilan

NOUS avons proposé DETEX, une technique de détection pour les défauts, en suivant les étapes de DECOR liées à la détection. DETEX permet de spécifier les règles de détection des défauts de code et de conception en utilisant un langage de haut niveau issu d'une analyse de domaine approfondie. À partir de ces spécifications, les algorithmes de détection sont directement générés grâce à la plate-forme 2DFW.

Nous avons spécifié 4 défauts de conception et les 15 défauts de code associés et généré automatiquement les algorithmes de détection en utilisant des gabarits. Ensuite, nous avons validé les algorithmes de détection générés en termes de précision et de rappel, pour la première fois, sur XERCES v2.7.0, un système logiciel libre ainsi qu'en terme de précision sur neuf autres systèmes libres afin de démontrer le passage à l'échelle des algorithmes de détection. Nous avons montré que les algorithmes de détection sont raisonnablement efficaces et précis et ont un bon rappel. Nous avons également appliqué les algorithmes de détection sur ECLIPSE v3.1.2, en mettant en valeur le problème de balance entre les nombres de classes suspectes et les précisions.

Ces expérimentations ont permis de montrer la pertinence des spécifications et l'utilité de la méthode DECOR via notre technique de detection DETEX. C'est la première fois qu'une validation d'une telle envergure a été réalisée pour évaluer la détection des défauts.

Dans ce travail, il est difficile de faire une comparaison avec les travaux précédents car ceux-ci n'expliquent pas concrètement leurs spécifications et leurs techniques pour la détection des défauts. Avec notre approche et sa validation en termes de précision et de rappel, nous définissons un point de repère pour de futures comparaisons quantitatives.

Le lecteur intéressé peut trouver le matériel de comparaison et de réplication incluant l'outil pour la détection des défauts, les fiches de règles et les résultats expérimentaux sur le site de DECOR [DECOR, 2006].

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Tableau 3.4 – Résultats de l'application des algorithmes de détection (*dans chaque colonne, la première ligne est le nombre de classes suspectes, la seconde ligne est le nombre de classes considérées comme étant des défauts de conception, la troisième ligne est la précision et la quatrième ligne donne le temps de détection. Les nombres entre parenthèses sont les pourcentages des classes reportées. F.D. = Functional Decomposition, S.C. = Spaghetti Code, and S.A.K. = Swiss Army Knife*).

	ArgoUML	Azureus	GanttProject	Log4J	Lucene	Nutch	PMD	QuickUML	Xerces v1.0.1	Xerces v2.7.0
Blob	29 (2.4%)	41 (2.8%)	10 (5.3%)	3 (1.6%)	3 (1.9%)	6 (2.9%)	4 (0.9%)	0 (0%)	10 (5.3%)	44 (8.6%)
	25 (2.0%)	38 (2.6%)	9 (4.8%)	3 (1.6%)	2 (1.3%)	4 (1.9%)	4 (0.9%)	0 (0%)	10 (5.3%)	39 (7.6%)
	86.2%	92.7%	90.0%	100%	66.7%	66.7%	100%	100%	100%	88.6%
	3.0s	6.4s	2.4s	1.3s	1.8s	3.6s	3.9s	0.4s	2.7s	2.4s
F.D.	37 (3.0%)	44 (3.0%)	15 (8.0%)	11 (5.8%)	1 (0.6%)	15 (7.2%)	13 (3.1%)	10 (7.0%)	4 (2.1%)	29 (5.6%)
	22 (1.8%)	17 (1.2%)	4 (2.1%)	6 (3.2%)	0 (0%)	3 (1.4%)	4 (0.9%)	3 (2.1%)	4 (2.1%)	15 (2.9%)
	59.5%	38.6%	26.7%	54.5%	0%	20.0%	30.8%	30.0%	100%	51.7%
	0.4s	0.5s	0.8s	0.05s	0.03s	0.05s	0.06s	0.02s	0.03s	0.16s
S.C.	44 (3.6%)	153 (15.6%)	14 (7.4%)	3 (1.6%)	8 (5.2%)	26 (12.6%)	9 (2.1%)	5 (3.5%)	25 (13.2%)	76 (14.8%)
	38 (3.1%)	125 (8.6%)	10 (5.3%)	2 (1.1%)	6 (3.9%)	22 (10.6%)	5 (1.2%)	0 (0%)	23 (12.2%)	46 (9.0%)
	86.4%	81.7%	71.4%	66.7%	75.0%	84.6%	55.6%	0%	92.0%	60.5%
	0.3s	2.9s	0.2s	0.08s	0.09s	0.1s	0.06s	0.03s	0.11s	0.2s
S.A.K.	108 (8.8%)	145 (10.0%)	8 (4.2%)	51 (27.0%)	9 (5.8%)	33 (15.9%)	13 (3.1%)	6 (4.2%)	12 (6.3%)	56 (10.9%)
	18 (1.5%)	33 (2.3%)	3 (1.6%)	33 (17.5%)	1 (0.6%)	13 (6.3%)	6 (1.4%)	1 (0.7%)	5 (2.6%)	23 (4.5%)
	16.6%	22.7%	37.5%	64.7%	11.1%	39.4%	46.1%	16.7%	41.7%	41.1%
	0.3s	0.13s	0.05s	0.02s	0.02s	0.02s	0.02s	0.02s	0.03s	0.05s
	62.2%	58.9%	56.4%	71.5%	38.2%	52.7%	58.1%	36.7%	83.4%	60.5%

□

Chapitre 4

Correction des défauts

Ce chapitre commence avec une description générale de notre approche pour la correction, qui est basée sur l'ARC et l'AFC, des méthodes algébriques pour identifier des groupes d'individus qui partagent des propriétés communes. Nous introduisons et illustrons à l'aide d'un exemple les notions de base sur lesquelles l'AFC et l'ARC reposent. Ensuite, nous explicitons le problème lié à la correction des défauts et proposons une technique de correction, appelée COREX, qui suit les étapes de DECOR liées à la correction. Nous illustrons ce problème ainsi que notre technique de correction en utilisant un système simple de gestion d'une bibliothèque, qui inclut un Blob. Enfin, nous présentons une étude empirique préliminaire sur la validité de l'approche afin d'évaluer COREX et ainsi de valider indirectement la méthode DECOR. Nous avons validé notre approche sur des instances du défaut de conception Blob détectées dans quatre systèmes libres différents. Les résultats montrent que l'ARC suggère un taux acceptable de restructurations pertinentes. Nous discutons également brièvement l'application de notre méthode sur d'autres défauts.

Sommaire

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

4.1 Approche liée à la correction des défauts

LA CORRECTION des défauts est une activité sujette à erreur et coûteuse en temps, d'où la nécessité d'avoir des techniques et outils automatisés [Sahraoui *et al.*, 2000]. Comme nous l'avons présenté dans les travaux reliés à la détection, il existe de nombreuses approches et outils pour détecter des défauts. Par contre, à notre connaissance, aucune approche ne tente de corriger les défauts détectés de manière semi- ou complètement automatique sans avoir à implémenter les règles de correction spécifiques à chaque défaut.

Nous avons appliqué la méthode DECOR pour proposer une nouvelle technique de correction pour les défauts de code et de conception, appelée COREX (*CORrection EXpert*) afin de suggérer des restructurations en utilisant l'Analyse Relationnelle de Concepts (ARC). L'ARC [Dao *et al.*, 2004 ; Huchard *et al.*, 2007] découle de l'Analyse Formelle de Concepts (AFC) [Ganter et Wille, 1999], qui est une méthode algébrique d'identification de groupes d'individus ayant des propriétés communes. L'ARC étend l'AFC avec le traitement des liens ou relations qui existent entre les individus.

La technique de correction COREX consiste à suggérer des restructurations appropriées pour corriger des défauts spécifiques en utilisant des refactorisations. En particulier, nous examinons les bénéfices de l'ARC pour la correction d'un défaut de conception très commun, le Blob [Brown *et al.*, 1998, pages 73–83]. Nous rappelons que le Blob révèle une conception (et une pensée) procédurale implémentée avec un langage de programmation orientée objet. Il se manifeste à travers une large classe contrôleur qui joue un rôle "divin" dans le système en monopolisant le traitement et qui est entourée par un certain nombre de petites classes de données fournissant beaucoup d'attributs mais peu ou aucune méthode. COREX peut s'appliquer également à d'autres types de défauts affectés par une faible cohésion et un fort couplage et listés dans la section 4.3.2.

L'ARC est particulièrement bien adaptée pour suggérer des restructurations afin de corriger les défauts car elle permet de prendre en compte les relations qui existent entre les entités d'un système telles que les invocations de méthodes ou les liens d'association. En effet, corriger un Blob revient à décomposer la classe Blob en plus petits ensembles

© 2008 Tous droits réservés.
http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

cohésifs en groupant les membres de la classe qui collaborent afin de réaliser une responsabilité spécifique. Dans notre contexte, les ensembles cohésifs correspondent aux groupes d'entités (des méthodes, par exemple) qui partagent des propriétés communes telles que l'accès à un attribut commun et des liens inter-individus tels que les invocations entre méthodes.

Avant de présenter les différentes étapes de la technique de correction COREX basée sur l'ARC, nous présentons dans la section suivante les notions de base de l'AFC et l'ARC.

4.2 Analyse de concepts

Nous présentons dans cette section seulement les éléments de base de l'AFC et de l'ARC et leurs définitions pour la bonne compréhension de notre approche. Pour de plus amples informations, le lecteur intéressé peut se référer à [Birkhoff, 1940 ; Barbut et Monjardet, 1970 ; Ganter et Wille, 1999 ; Huchard *et al.*, 2007].

4.2.1 Analyse formelle de concepts

L'AFC [Ganter et Wille, 1999] est une méthode algébrique d'identification de groupes d'individus (ou objets formels) ayant des propriétés communes (ou attributs formels)[1]. Les ensembles d'individus et de propriétés mutuellement correspondants sont appelés *concepts formels*. Ces concepts sont organisés en une hiérarchie, dite *treillis de concepts*, par le biais de la relation d'ordre partielle qui repose sur la relation d'inclusion ensembliste entre groupes d'individus et de propriétés. L'AFC puise ses bases dans les travaux de Birkhoff [1940] et Barbut *et al.* [1970] sur la théorie des treillis et des ordres et a été constituée en théorie à part entière par Rudolph Wille dès le début des années 80.

4.2.1.1 Contexte binaire

Dans l'AFC, les données sont représentées sous la forme d'un tableau booléen à deux dimensions, appelé *contexte (formel) binaire*, définissant la relation d'incidence entre deux ensembles finis d'individus et de propriétés.

Un contexte binaire se décrit formellement comme suit :

Un *contexte binaire* est un triplet $\mathcal{K} = (\mathcal{O}, \mathcal{A}, \mathcal{I})$ où :
- $\mathcal{O}$ est l'ensemble des individus formels ;
- $\mathcal{A}$ est l'ensemble des propriétés formelles ;
- $\mathcal{I}$ est une relation binaire entre les éléments de $\mathcal{O}$ et $\mathcal{A}$ telle que $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{A}$ indiquant pour chaque individu les propriétés qui lui sont associées.

[1] Nous utiliserons dans la suite *individus* au lieu d'*objets* et *propriétés* au lieu d'*attributs* pour éviter toute confusion avec les objets et attributs orientés objet.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Supposons les résultats issus d'un questionnaire concernant le genre et l'âge d'un groupe de personnes ; ces résultats sont reportés dans le tableau 4.1. Le contexte binaire donné dans le tableau 4.2 correspond au tableau de données 4.1 et définit le format dans lequel les données sont présentées dans l'AFC. Nous utilisons cet exemple de données et de contexte binaire dans la suite pour illustrer les notions fondamentales de l'AFC et l'ARC.

Tableau 4.1 – Exemple d'un tableau de données.

	Genre	Âge
Jean	Masc	18
Jane	Fem	19
Marc	Masc	16
Paul	Masc	27
Lise	Fem	13
Anne	Fem	25

□

Tableau 4.2 – Exemple d'un contexte binaire.

	Genre		Âge	
	Masc	Fem	Jeune ≤ 20	Adulte > 20
Jean	×		×	
Jane		×	×	
Marc	×		×	
Paul	×			×
Lise		×	×	
Anne		×		×

□

4.2.1.2 Scaling conceptuel

Le mécanisme qui a permis de passer du tableau de données 4.1 au contexte binaire du tableau 4.2 est appelé *scaling* [2] *conceptuel*.

Selon la définition donnée par Ganter et Wille [Ganter et Wille, 1999], le *scaling conceptuel* correspond au mécanisme qui permet de transformer des contextes multi-valués en contextes binaires équivalents.

En effet, le tableau 4.1 appelé *contexte multi-valué* contient plusieurs types de valeurs pour le genre et l'âge. La propriété multi-valuée Âge dont les valeurs sont {18, 19, 16, 27, 13, 25} a été codée par des prédicats unaires tels que 'Jeune' et 'Adulte' qui expriment les âges inférieurs et strictement supérieurs à 20 tels que : Jeune ≤ 20, Adulte > 20, comme indiqué dans le tableau 4.2.

[2] *Les termes français échelonnage ou étalonnage peuvent être utilisés à place de scaling, mais comme il n'y a pas de consensus, nous utiliserons simplement le terme scaling.*

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

4.2.1.3 Correspondance de Galois

L'AFC permet la génération d'une structure de treillis que la relation binaire entre les deux ensembles, individus et propriétés, induit à travers la correspondance de Galois sous-jacente telle que définie comme suit :

Soient $f : U \rightarrow V$ et $g : V \rightarrow U$ des fonctions définies sur deux ensembles ordonnés $(U, \leq_U)$ et $(V, \leq_V)$.

(f, g) est une correspondance de Galois si pour tout $u, u_1, u_2 \in U$ et pour tout $v, v_1, v_2 \in V$:

- $u_1 \leq_U u_2 \Longrightarrow f(u_2) \leq_V f(u_1)$ (f est décroissante)
- $v_1 \leq_V v_2 \Longrightarrow g(v_2) \leq_U g(v_1)$ (g est décroissante)
- $u \leq_U g(f(u))$ et $v \leq_V f(g(v))$ (les composés sont extensifs)

Dans un contexte binaire, la correspondance de Galois est définie comme suit :

Etant donné le contexte binaire $\mathcal{K} = (\mathcal{O}, \mathcal{A}, \mathcal{I})$, nous définissons les deux fonctions f et g ainsi : soit $X \subseteq \mathcal{O}$,

- $f : 2^{\mathcal{O}} \rightarrow 2^{\mathcal{A}}$, $f(X) = X' = \{a \in \mathcal{A} | \forall o \in X, (o, a) \in \mathcal{I}\}$

et de manière duale, soit $Y \subseteq \mathcal{A}$,

- $g : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{O}}$, $g(Y) = Y' = \{o \in \mathcal{O} | \forall a \in Y, (o, a) \in \mathcal{I}\}$

A partir de ces deux fonctions, il est possible de dériver les règles suivantes [Ganter et Wille, 1997] : pour tout $X_1, X_2, X \subseteq \mathcal{O}$ et pour tout $Y_1, Y_2, Y \subseteq \mathcal{A}$,

1) $X_1 \subseteq X_2 \Rightarrow f(X_2) \subseteq f(X_1)$ 2) $X \subseteq g(f(X))$ et $f(X) = f(g(f(X)))$
1') $Y_1 \subseteq Y_2 \Rightarrow g(Y_2) \subseteq g(Y_1)$ 2') $Y \subseteq f(g(Y))$ et $g(Y) = g(f(g(Y)))$

$$X \subseteq g(Y) \Leftrightarrow Y \subseteq f(X)$$

Ainsi, il existe donc une correspondance de Galois entre les deux ensembles $\mathcal{O}$ et $\mathcal{A}$ et donc un isomorphisme dual.

De même, la paire de fonctions (f, g), résumant les relations entre les individus et les propriétés dans le contexte, définit une correspondance de Galois entre les deux ensembles ordonnés $(2^{\mathcal{O}}, \subseteq)$ et $(2^{\mathcal{A}}, \subseteq)$.

Par exemple, dans le contexte binaire donné dans le tableau 4.2, la correspondance de Galois s'illustre comme suit :

$$\{\texttt{Jane, Lise}\}' = \{\texttt{Fem, Jeune}\}$$
$$\{\texttt{Masc, Jeune}\}' = \{\texttt{Jean, Marc}\}$$

Les méthodes d'analyse de l'AFC permettent l'extraction de tous les groupes d'individus ayant des propriétés communes comme indiqué par la correspondance de Galois. Ces groupes sont appelés des *concepts formels* et sont organisés en une hiérarchie, appelée *treillis de concepts*.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

4.2.1.4 Concept formel

D'un point de vue philosophique, un *concept* est une unité de pensée constituée de deux parties : l'*intension* et l'*extension*. L'extension couvre tous les individus appartenant à ce concept et l'intension comprend toutes les propriétés valides pour tous les individus [Wagner, 1973].

De la même façon, un *concept formel* est constitué de deux parties se caractérisant mutuellement : l'*extension* qui contient les individus appartenant au concept et l'*intension* qui contient les propriétés partagées par les individus.

Un concept formel se décrit formellement comme suit :

Un *concept formel* d'un *contexte* $\mathcal{K} = (\mathcal{O}, \mathcal{A}, \mathcal{I})$, est une paire (X, Y) où :
- $X \in 2^{\mathcal{O}}$ est appelé l'*extension* ;
- $Y \in 2^{\mathcal{A}}$ est appelé l'*intension*.

Une paire (X,Y) est un concept formel de $\mathcal{K}$ si et seulement si :

$$X \subseteq \mathcal{O}, Y \subseteq \mathcal{A}, X = Y' \text{ et } Y = X'$$

Les concepts d'un contexte donné sont ordonnés naturellement par une relation de *sous-concept* et de *super-concept* [Ganter et Wille, 1997] qui repose sur la relation d'inclusion entre sous-ensembles d'individus et de propriétés :

$$(X_1, Y_1) \leq (X_2, Y_2) :\Leftrightarrow X_1 \subseteq X_2 (\Leftrightarrow Y_1 \subseteq Y_2)$$

4.2.1.5 Treillis de concepts

L'ensemble de tous les concepts formels de $\mathcal{K} = (\mathcal{O}, \mathcal{A}, \mathcal{I})$, ordonné par la relation de super-concept (ou sous-concept) est désigné par $\mathcal{L} = \langle \mathcal{C}_{\mathcal{K}}, \leq_{\mathcal{K}} \rangle$ et est appelé un *treillis de concepts* de $\mathcal{K}$.

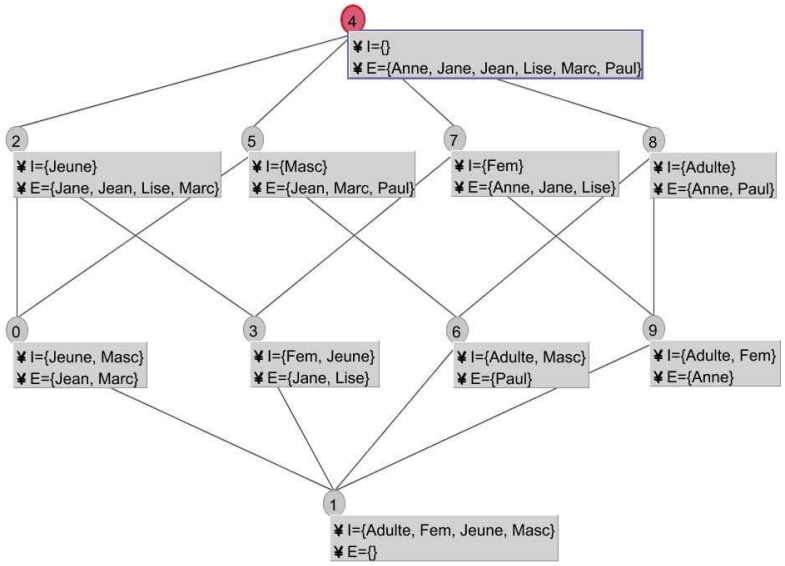
Le treillis de la figure 4.1 correspond au contexte binaire donné dans le tableau 4.2. Le nœud 2 du treillis contenant le couple ({`Jean,Jane,Marc,Lise`},{`Jeune`}) correspond au concept $c2$ du treillis : les individus `Jean`, `Jane`, `Marc` et `Lise` possèdent la propriété `Jeune` qui, à son tour, caractérise exclusivement ces quatre individus. La figure 4.1 présente un treillis avec *étiquetage complet*, c'est-à-dire que tous les individus et toutes les propriétés qui définissent un concept sont explicitement indiqués dans l'intension et l'extension du concept. La figure 4.2 présente un treillis avec *étiquetage réduit*, c'est-à-dire qu'un concept donné est défini par l'ensemble des individus situés en-dessous et des propriétés situées au-dessus.

En effet, l'étiquetage réduit (en anglais, *reduced labeling*) est une manière de représenter un treillis de concepts et consiste à afficher un individu (respectivement une propriété) une seule fois dans le treillis au niveau du noeud où il apparaît pour la première fois. Ainsi, l'extension complète (resp. l'intension) d'un concept est obtenue en cumulant sans

© 2008 Tous droits réservés.
http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 4.1 – Exemple d'un treillis de concepts.



□

répétition tous les individus (resp. les propriétés) affichés en-dessous (resp. au-dessus) de ce concept.

Dans la section suivante, nous présentons les notions fondamentales de l'ARC qui est une extension de l'AFC.

4.2.2 Analyse relationnelle de concepts

L'ARC [Dao *et al.*, 2004 ; Huchard *et al.*, 2007] est une approche permettant l'extraction de concepts formels à partir d'ensembles d'individus décrits par des propriétés propres et des liens inter-individus. Les concepts formés sont dits *concepts relationnels* car les intensions qu'ils renferment font référence à d'autres concepts.

4.2.2.1 Famille de contextes relationnels

Les données de l'ARC sont organisées au sein d'une structure, appelée *Famille de Contextes Relationnels* (FCR), décrite comme suit :

Une FCR est une paire (K, R) où :

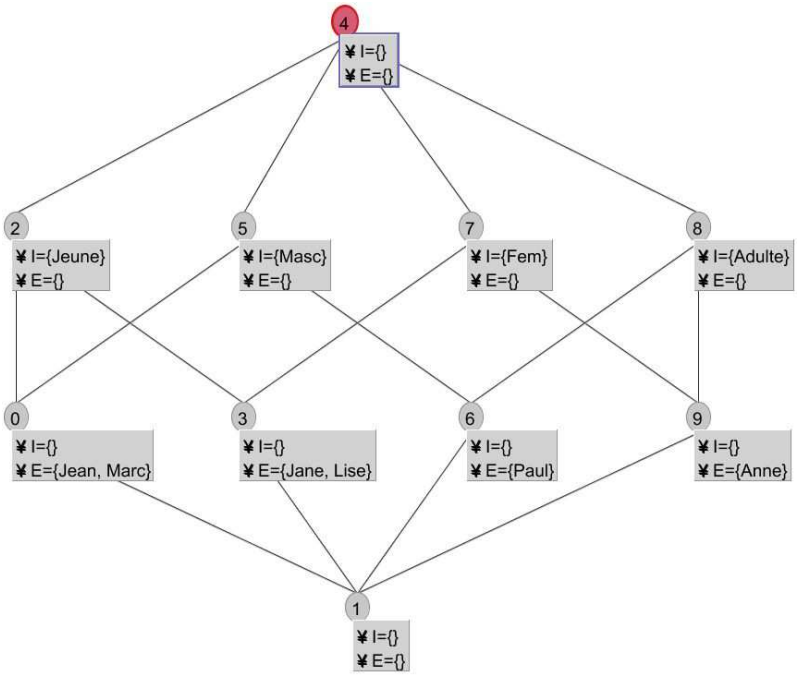
- K est un ensemble de contextes binaires $\mathcal{K}_i = (\mathcal{O}_i, \mathcal{A}_i, \mathcal{I}_i)$;
- R est un ensemble de relations binaires $r_k \subseteq \mathcal{O}_i \times \mathcal{O}_j$, où $\mathcal{O}_i$ et $\mathcal{O}_j$ sont des ensembles d'individus de $\mathcal{K}_i$ et $\mathcal{K}_j$, appelés respectivement domaine et co-domaine de r_k.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 4.2 – Exemple d'un treillis de concepts avec étiquetage réduit.



□

Considérons la relation *ami* qui définit une relation d'amitié entre différentes personnes. Le tableau 4.3 décrit les liens d'amitiés entre les individus de notre exemple vérifiant la relation *ami*. La relation binaire correspondante est donnée dans le tableau 4.4. Le contexte binaire du tableau 4.2 et la relation binaire du tableau 4.4 forment un échantillon FCR.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Tableau 4.3 – Table représentant des liens inter-individus.

	ami
Jean	{Paul}
Jane	
Marc	{Jane}
Paul	
Lise	{Jane}
Anne	{Jean, Paul}

□

Tableau 4.4 – Exemple d'une relation binaire.

	Jean	Jane	Marc	Paul	Lise	Anne
Jean				×		
Jane						
Marc		×				
Paul						
Lise		×				
Anne	×			×		

□

4.2.2.2 Scaling relationnel

Comme nous l'avons précédemment présenté, le *scaling conceptuel* est un mécanisme qui permet de convertir les contextes multi-valués en contextes binaires. Quant au *scaling relationnel*, il s'agit d'un mécanisme similaire au scaling conceptuel qui consiste à intégrer les liens inter-individus sous la forme de nouvelles propriétés binaires afin de dériver des structures conceptuelles où les concepts peuvent dépendre d'autres concepts [Rouane-Hacene, 2007].

Par définition, le scaling relationnel permet d'ajouter au contexte $\mathcal{K}_i$ de nouvelles propriétés dites *relationnelles* et créées à partir de contextes multi-valués. Ces propriétés se présentent sous la forme $r : c$, où r est une relation donnée telle que $r : O_i \rightarrow 2^{O_j}$ et c est un concept formel sur $\mathcal{K}_j$.

Schéma de scaling. Étant donnée la relation $r : O_i \rightarrow 2^{O_j}$ et c est un concept formel sur $\mathcal{K}_j$, il existe plusieurs façons d'associer les propriétés relationnelles de type $r : c$ à l'individu o appartenant au contexte $\mathcal{K}_i$. Ces différentes façons, appelées *schémas de scaling*, dépendent du rapport entre l'ensemble $r(o)$ des individus auxquels l'individu o est lié et l'extension du concept $c = (X, Y)$. Le rapport peut être, par exemple, une inclusion telle que $r(o) \subseteq X$ et dans ce cas, on parle de *schéma de scaling universel strict* ou une intersection non vide telle que $r(o) \cap X \neq \emptyset$, appelé *schéma de scaling existentiel*.

Dans l'étude présente, comme dans la vaste majorité des applications du génie logiciel de l'ARC, nous avons utilisé le schéma de scaling existentiel.

Opérateur de scaling existentiel. L'opérateur de scaling existentiel est décrit formellement de la manière suivante :

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Étant donné un contexte $\mathcal{K}_i = (\mathcal{O}_i, \mathcal{A}_i, \mathcal{I}_i)$, le treillis de concepts $\mathcal{L}_j$ associé au contexte $\mathcal{K}_j = (\mathcal{O}_j, \mathcal{A}_j, \mathcal{I}_j)$ et la relation $r \subseteq \mathcal{O}_i \times \mathcal{O}_j$,
l'*opérateur de scaling existentiel* $sc_{\exists}^{(r,\mathcal{L}_j)}$ est défini par :

$$sc_{\exists}^{(r,\mathcal{L}_j)} : K \rightarrow K$$

$$sc_{\exists}^{(r,\mathcal{L}_j)}(\mathcal{K}_i) = (\mathcal{O}_i^{(r,\mathcal{L}_j)}, \mathcal{A}_i^{(r,\mathcal{L}_j)}, \mathcal{I}_i^{(r,\mathcal{L}_j)})$$

avec :
- $\mathcal{O}_i^{(r,\mathcal{L}_j)} = \mathcal{O}_i$;
- $\mathcal{A}_i^{(r,\mathcal{L}_j)} = \mathcal{A}_i \cup \{\exists r : c | c \in \mathcal{L}_j\}$;
- $\mathcal{I}_i^{(r,\mathcal{L}_j)} = \mathcal{I}_i \cup \{(o, r : c) | o \in \mathcal{O}_i, c = (X, Y) \in \mathcal{L}_j, r(o) \cap X \neq \emptyset\}$.

Le tableau 4.5 présente le résultat du scaling de la relation *ami* qui étend le contexte donné dans le tableau 4.2 avec les nouvelles propriétés relationnelles `ami:c3`, `ami:c6`, `ami:c7`, `ami:c8`, `ami:c14`. Les concepts $\{c3, c6, c7, c8, c14\}$ correspondent aux concepts du treillis dérivé à partir du contexte initial du tableau 4.2 et présenté dans la figure 4.3.

Tableau 4.5 – Exemple d'un scaling relationnel.

	ami : $c3$	ami : $c6$	ami : $c7$	ami : $c8$	ami : $c14$
Jean		×			
Jane					
Marc	×				
Paul					
Lise	×				
Anne		×		×	×

□

4.2.2.3 Famille de treillis relationnels

Une fois que toutes les relations ont été traitées par le scaling relationnel, les contextes binaires de la FCR sont traités à leur tour par les algorithmes classiques de dérivation de structures conceptuelles de l'AFC. Ainsi, à partir d'une FCR, nous obtenons un ensemble de treillis, un par contexte, appelé *Famille de Treillis Relationnels* (FTR).

Une FTR est définie comme l'ensemble des treillis dont les concepts reflètent conjointement toutes les propriétés partagées et toutes les relations parmi les individus d'une FCR.

Sa construction est un processus itératif car le mécanisme de scaling modifie les contextes ainsi que les treillis correspondants, lesquels à leur tour peuvent nécessiter un nouveau scaling qui reflètent les concepts nouvellement formés et la relation partagée qu'ils

© 2008 Tous droits réservés.

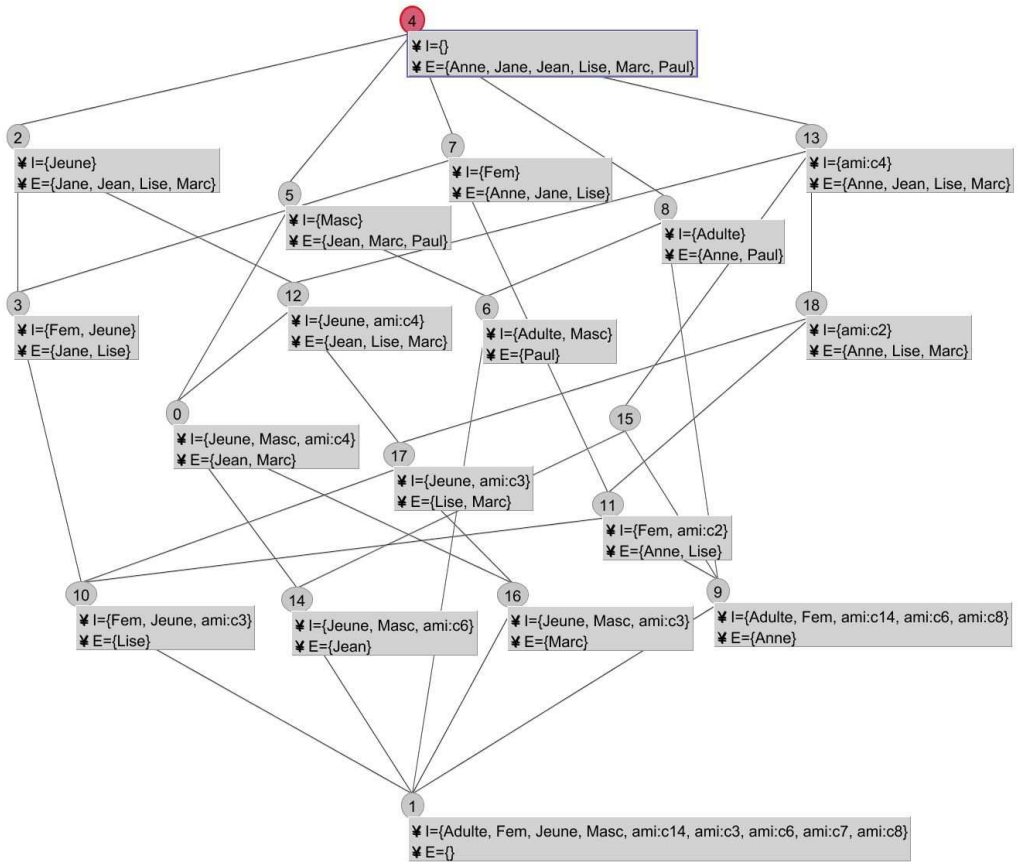
http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

provoquent. Les itérations s'arrêtent une fois qu'un point fixe est atteint, c'est-à-dire lorsqu'un autre scaling laisse tous les treillis dans une FTR inchangée.

Le treillis relationnel correspondant à notre exemple est présenté sur la figure 4.3. L'individu `Anne` dans le concept $c9$ a une relation *ami* avec les individus `Jean` et `Paul` qui apparaissent respectivement dans l'extension des concepts $c14$ et $c6$ (ainsi que dans $c8$). Les concepts $c10$ et $c16$ qui représentent respectivement les individus `Lise` et `Marc` ont été assignés à la propriété relationnelle `ami:c3`, ce qui signifie que `Marc` et `Lise` ont une relation commune *ami* avec les individus situés dans l'extension du concept $c3$, en l'occurrence `Jane`. Leur relation avec l'individu `Jane` a été révélé lors du scaling et le nouveau concept $c17$ a émergé afin de représenter cette propriété commune. Nous pouvons observer grâce à ce nouveau concept que des individus du même âge entretiennent une relation d'amitié, chose que nous n'aurions pas pu voir autrement.

FIG. 4.3 – Treillis final de l'exemple.



□

Le treillis final de la figure 4.3 est différent du treillis initial de la figure 4.1 à cause de l'information relationnelle insérée dans le contexte initial. En effet, les individus sont assignés aux propriétés relationnelles qui mènent au partage d'un nombre plus important

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

de propriétés parmi ces individus. En factorisant les nouvelles propriétés dans les intensions des concepts, les liens entre les individus sont remontés vers le niveau concept, fournissant les relations entre les concepts et créant éventuellement de nouveaux concepts.

Toutes les notions de l'AFC et de l'ARC qui viennent d'être présentées et définies vont permettre la bonne compréhension de l'approche que nous proposons pour la correction des défauts. Dans la section suivante, nous explicitons le problème que soulève la correction des défauts avant de présenter notre approche dans la section qui suit.

4.3 Problème de correction des défauts

Le problème lié à la correction des défauts consiste à suggérer des restructurations afin de supprimer les défauts tout en améliorant la structure et la qualité des systèmes analysés. Cependant, ce problème n'est pas simple car la correction des défauts a un impact sur plusieurs critères de qualité et doit tenir compte de plusieurs facteurs. Nous nous limitons donc, dans le cadre de ce travail de thèse, à améliorer un sous-ensemble de critères de qualité non respectés par un sous-ensemble de défauts.

4.3.1 Critères de qualité

Les défauts sont les résultats de *mauvaises* pratiques qui transgressent les *bons* principes orientés objet. Le processus de correction vise à satisfaire ces principes. Nous nous limitons à l'évaluation du *couplage* et de la *cohésion*, qui sont parmi les caractéristiques de qualité logicielle les plus largement reconnues, clés pour la maintenance [Bart Du Bois et Demeyer, 2004], telles que définies ci-dessous :

Cohésion. La cohésion d'une classe reflète *à quel point les méthodes sont étroitement liées aux attributs et aux méthodes de la classe* et est typiquement mesurée par la métrique LCOM (Lack of COhesion Metric : entre 0 et 1) laquelle utilise *le nombre d'ensembles disjoints d'attributs et de méthodes* [Fenton et Pfleeger, 1997].

Une faible valeur LCOM caractérise une classe cohésive alors qu'une valeur proche de 1 indique un manque de cohésion et suggère que la classe devrait être plutôt décomposée en ensembles cohésifs. Cette métrique se calcule sur une classe.

Couplage. Le couplage d'une classe par rapport au reste d'un système est défini comme *le degré de dépendance des services fournis aux autres classes* [Fenton et Pfleeger, 1997]. Il est mesuré par la métrique CBO (Coupling Between Objects) [Chidamber et Kemerer, 1994] qui correspond au *nombre de classes auxquelles une classe est couplée*.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Une valeur CBO nulle caractérise une classe complètement indépendante du reste du système alors qu'une valeur CBO élevée indique qu'une classe est fortement couplée et donc dépendante du reste du système. Cette métrique se calcule sur plusieurs classes.

Un système bien conçu exhibe une *forte* cohésion et un *faible* couplage, mais il est largement reconnu que ces critères sont concurrents et donc un compromis est généralement recherché.

4.3.2 Application sur d'autres défauts

Nous choisissons d'illustrer notre approche avec le Blob car il a un impact négatif à la fois sur la cohésion et le couplage : les Blobs présentent une faible cohésion et un fort couplage. De plus, il s'agit d'un défaut fréquent dans les systèmes orientés objet. Par exemple, une précédente étude a révélé 1 146 Blobs dans l'environnement de développement ECLIPSE, même si celui-ci est reconnu pour la qualité de sa conception.

Pourtant, nous avons trouvé qu'un bon nombre d'autres défauts sont affectés par une faible cohésion et un fort couplage, par exemple, Divergent Change [Fowler, 1999, page 79], Feature Envy [Fowler, 1999, page 80], Inappropriate Intimacy [Fowler, 1999, page 85], Lazy Class [Fowler, 1999, page 83], Shotgun Surgery [Fowler, 1999, page 80] et Swiss Army Knife [Brown *et al.*, 1998, page 197]. Ainsi, notre approche pourrait être adaptée à ces défauts.

4.3.3 Exemple illustratif

Notre exemple illustratif présenté sur la figure 4.4 est inspiré d'un système simple de gestion d'une bibliothèque décrit dans [Brown *et al.*, 1998], lequel inclut un Blob. La large classe contrôleur est la classe `Library_Main_Control` qui accède aux attributs de deux classes de données associées `Book` et `Catalog`.

4.3.4 Processus de refactorisation

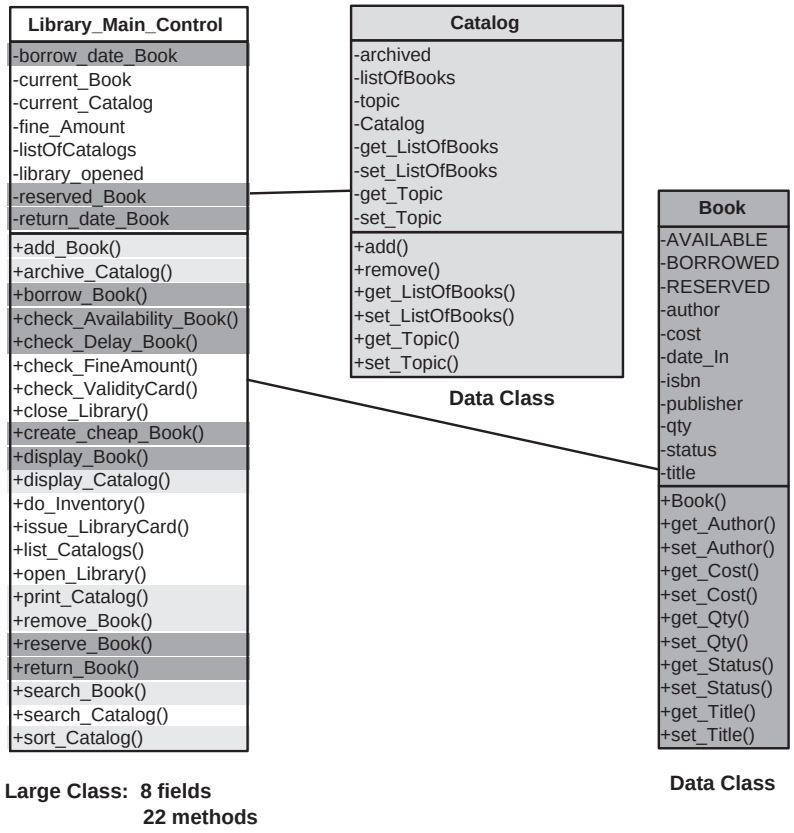
Corriger un Blob consiste à déplacer les membres de la large classe contrôleur vers ces classes de données ou vers de nouvelles classes conçues spécialement. Concernant la classe `Library_Main_Control`, nous remarquons que les méthodes et attributs associés à `Book` ou `Catalog` peuvent être déplacés dans ces classes de données respectives. Comme résultat, les classes de données gagnent plus de comportement alors que la large classe devient moins complexe. Cependant, le processus pour choisir et appliquer la restructuration est long et fastidieux : les ingénieurs du logiciel ont besoin d'examiner toutes les méthodes et attributs de la large classe afin d'identifier les sous-ensembles de celle-ci qui forment des ensembles cohésifs et cohérents et déterminer vers quelles classes de données il faut les déplacer. Pourtant, il s'agit d'un effort nécessaire car le résultat du processus peut considérablement améliorer la structure du système.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 4.4 – Diagramme de classes du Blob lié à la gestion d'une bibliothèque (*les attributs et méthodes en gris foncé et en gris clair de la classe* `Library_Main_Control` *sont des membres de classe qui sont associés respectivement aux classes de données* `Book` *et* `Catalog`).



□

4.3.5 Solution au problème lié à la correction

Notre intuition est que les défauts résultant d'un fort couplage et d'une faible cohésion peuvent être améliorés en redistribuant les membres de classe (c'est-à-dire les attributs et les méthodes) parmi les classes existantes ou nouvellement créées afin d'augmenter la cohésion et/ou diminuer le couplage. L'ARC fournit un cadre approprié à la redistribution car elle permet de découvrir des ensembles d'individus fortement reliés à des propriétés partagées et des liens inter-individus et donc supporte la recherche de sous-ensembles cohésifs de membres de classe. Par ailleurs, la structure du treillis permet une navigation et une recherche faciles ainsi qu'une représentation optimale de l'information comparable au besoin de l'orienté objet classique de factorisation maximale puisque chaque individu et chaque propriété sont représentés canoniquement par un concept unique.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Par exemple, dans le cas du Blob, la solution pour le corriger consiste à suggérer un ensemble de refactorisations pour décomposer la large classe du Blob en autant de classes possibles qu'il y a d'ensembles cohésifs et à fusionner le contenu des classes de données associées avec les nouvelles classes lorsque celles-ci sont fortement couplées.

4.4 COREX : technique de correction

Nous rappelons qu'à l'issue de la première partie de la méthode DECOR, correspondant à la détection des défauts, nous obtenons une liste d'entités identifiées comme étant des défauts. Dans la seconde partie de la méthode DECOR, correspondant à la correction, ces entités sont ensuite analysées afin de suggérer des refactorisations pour corriger ces défauts.

Dans cette section, nous présentons la technique de correction COREX dont les étapes suivent les étapes de la méthode DECOR liées à la correction. Nous illustrons la technique en utilisant l'exemple présenté dans la précédente section.

La figure 4.5 décrit les quatre étapes de la technique de correction COREX pour l'identification des restructurations afin de corriger les défauts en général et le Blob en particulier. Sur la figure, nous mettons en valeur les étapes, entrées et sorties qui sont spécifiques à COREX. Cette approche se décompose en quatre étapes automatiques :

1. **étape 1. Analyse des entités** : cette étape consiste à identifier toutes les entités liées au défaut identifié au niveau d'une entité ;
2. **étape 2. Extraction de la FCR** : à partir d'un modèle du système à analyser et des classes ayant des défauts, une FCR encodant les relations entre les membres de classe est automatiquement extraite ;
3. **étape 3. Dérivation du treillis** : la FCR obtenue est introduite dans un moteur ARC qui dérive la FTR correspondante ;
4. **étape 4. Exploration** : les concepts découverts sont explorés en utilisant des algorithmes simples qui consistent à appliquer un ensemble de règles de refactorisations afin d'identifier les ensembles cohésifs de méthodes et d'attributs. À l'issue de cette étape, nous obtenons un ensemble de restructurations suggérées par l'approche afin de corriger le défaut.

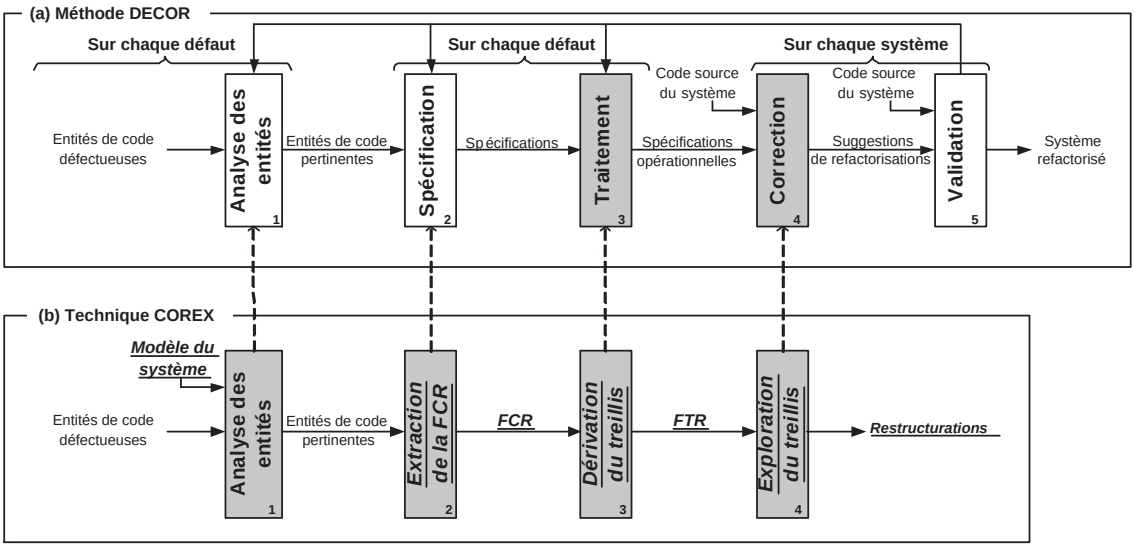
Dans la suite, nous détaillons les quatre étapes de la technique de détection COREX.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

FIG. 4.5 – (a) La méthode DECOR. (b) La technique de correction COREX (*les étapes, entrées et sorties en gras, italiques et soulignés sont spécifiques à* COREX *en comparaison avec* DECOR).



□

4.4.1 Étape 1. Analyse des entités

À l'issue de la détection, un fichier textuel liste les entités qui participent à la présence d'un défaut. Ce sont ces entités qui sont extraites de manière automatique dans le modèle du système analysé et qui vont être étudiées dans les étapes suivantes de la technique de correction. Ainsi, les méthodes et les attributs de la large classe sont fournis par le modèle du système qui représente les classes qui participent au Blob.

4.4.2 Étape 2. Extraction de la FCR

Pour corriger les défauts, nous avons besoin d'identifier les ensembles cohésifs de méthodes selon le mode d'accès des attributs et les invocations entre méthodes.

Ainsi, nous définissons des contextes avec comme individus les méthodes de la large classe et comme propriétés ses attributs. La relation d'incidence représente l'accès aux attributs par les méthodes en mode lecture ou écriture. Un tel contexte permet d'identifier les ensembles hautement cohésifs.

Nous considérons également les invocations de méthodes en les encodant par une relation dédiée inter-individus désignée par *call* afin de pouvoir réduire le couplage. Grouper des méthodes qui s'invoquent ou qui appellent un même ensemble d'attributs dans

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

une même classe décroît inévitablement le couplage.

La figure 4.6 présente un échantillon de la FCR constitué du contexte encodant l'accès des attributs par les méthodes et de la relation binaire *call* qui lie les méthodes du Blob entre elles.

Le tableau à gauche de la figure 4.6 illustre un contexte binaire dérivé à partir de la classe `Library_Main_Control` où les individus sont les méthodes du Blob alors que les propriétés sont les méthodes et les attributs accédés. Les préfixes `R-` et `W-` qui apparaissent dans les noms d'attributs spécifient le mode d'accès en lecture (*Read*) et écriture (*Write*), respectivement.

Le tableau à droite correspond à la relation binaire *call* qui lie les méthodes du Blob entre elles. Cette relation binaire a été obtenue après un scaling relationnel appliqué au contexte multi-valué correspondant au tableau 4.6. Les deux méthodes `borrow_Book()` et `reserve_Book()` appellent `check_Availability_Book()`. Affecter les deux premières méthodes à la même classe va permettre de réduire le couplage.

FIG. 4.6 – **Gauche :** Contexte des méthodes : accès des attributs par les méthodes. **Droite :** Relation binaire *call* entre les méthodes.

	'add_Book()'	'borrow_Book()'	'check_Availability_Book()'	'check_FineAmount()'	'close_Library()'	'issue_LibraryCard()'	'open_Library()'	'remove_Book()'	'reserve_Book()'	'return_Book()'	'search_Book()'	'sort_Catalog()'	R-current_Book	R-fine_Amount	W-borrow_date_Book	W-library_Opened	W-reserved_Book	W-return_date_Book
add_Book()	×												×					
borrow_Book()		×											×		×			×
check_Availability_Book()			×										×					
check_FineAmount()				×										×				
close_Library()					×											×		
issue_LibraryCard()						×												
open_Library()							×									×		
remove_Book()								×										
reserve_Book()									×								×	
return_Book()										×					×			×
search_Book()											×							
sort_Catalog()												×						

	add_Book()	check_Availability_Book()	check_FineAmount()	remove_Book()
		×		
			×	
		×		
	×			×

□

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Tableau 4.6 – Table des liens inter-individus correspondant à la relation *call* (*Les individus qui n'ont pas de relation avec d'autres individus ont été volontairement omis.*) .

	call
borrow_Book()	*check_Availability_Book()*
issue_LibraryCard()	*check_FineAmount()*
reserve_Book()	*check_Availability_Book()*
sort_Catalog()	*add_Book(),remove_Book()*

□

Des propriétés formelles correspondant à une chaîne de caractères sont dérivées à partir des noms des méthodes et ajoutées au contexte des méthodes comme dans le contexte binaire de la figure 4.6 à gauche. Ces propriétés permettent l'émergence d'un concept unique pour chaque méthode, appelé *concept méthode*[3], dans le treillis correspondant. Le concept méthode aide à préserver les invocations une-à-une entre méthodes car ces informations peuvent être perdues durant l'étape de scaling qui a pour but d'intégrer la relation *call* en substituant des invocations une-à-plusieurs à celles de type une-à-une.

Par exemple, supposons deux méthodes $m1$ et $m2$ appartenant à un même concept $c1$ et, dans le concept $c2$, une troisième méthode $m3$ qui appelle uniquement $m1$. Lors du scaling relationnel, nous obtenons au niveau de $c2$ une propriété relationnelle $call : c1$; par contre, nous ne savons pas si $m3$ appelle $m1$ et/ou $m2$. L'introduction d'un *concept méthode* pour chaque méthode évite de perdre ce genre d'information car il est alors possible de retracer les invocations exactes au niveau du treillis relationnel final.

4.4.3 Étape 3. Dérivation du treillis

La figure 4.7 représente le treillis de concepts obtenu via le moteur ARC à partir du contexte donné à la figure 4.6. Les concepts formels du treillis impliquent naturellement la cohésion car leurs extensions et intensions forment des *ensembles maximaux* d'individus partageant un ensemble maximal de propriétés. Ainsi, ils représentent les opportunités de refactorisation du défaut de conception. En effet, les concepts tels que $c9$ ({open_Library(), close_Library()},{W-library_opened}) exhibent un groupe de méthodes utilisant les mêmes ensembles d'attributs et les attributs utilisés par les ensembles cohésifs de méthodes. Ces concepts sont considérés comme classes candidates car ils sont cohésifs. En plus, les concepts tels que $c3$ et $c12$ mettent en valeur les sous-ensembles de méthodes cohésives, car les méthodes appelant le même ensemble d'autres méthodes sont fortement cohésives. Une troisième catégorie de concepts tels que $c9$ et $c13$ représentent la *relation d'utilisation* entre les méthodes de la large classe et les classes de données associées. L'étude de ces concepts permet d'évaluer le couplage entre la large

[3]La plus petite extension dans le treillis contient cette méthode.

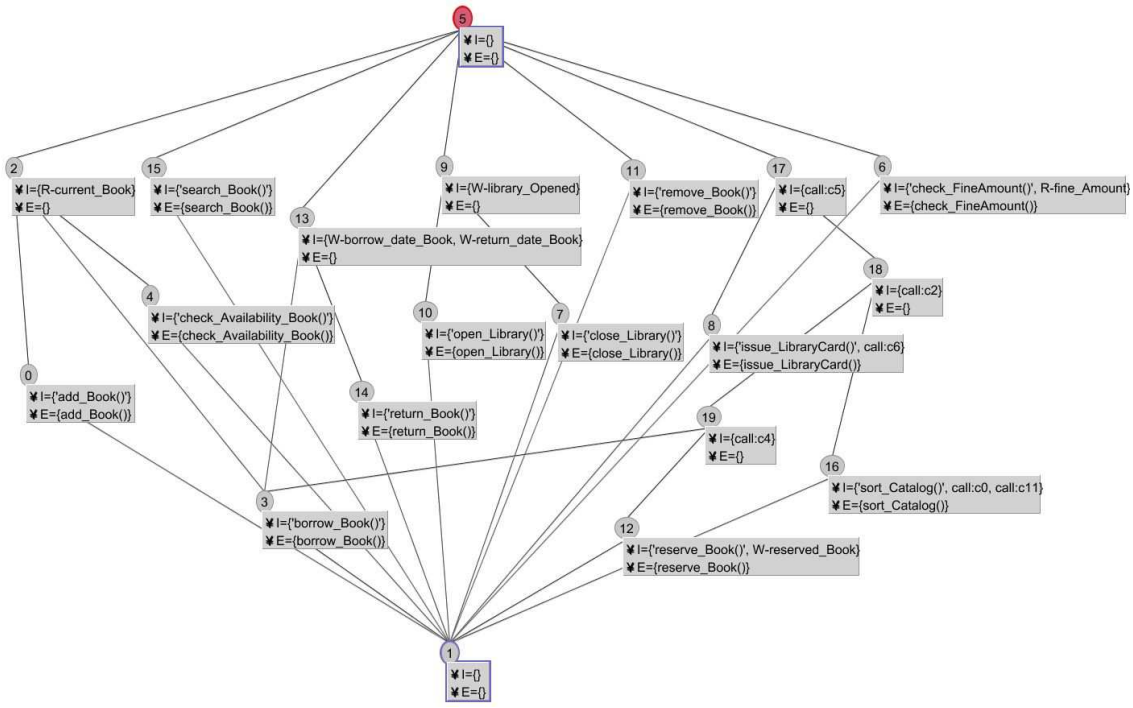
© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

classe et les classes de données. Ainsi, nous pouvons identifier quelles méthodes et quels attributs de la large classe devraient être déplacés vers les classes de données.

FIG. 4.7 – Treillis associé au contexte de la figure 4.6.



□

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

4.4.4 Étape 4. Exploration du treillis

Le treillis dérivé dans l'étape précédente et inclus dans une FTR est utilisé pour suggérer les restructurations. Plus spécifiquement, nous appliquons des algorithmes cherchant des concepts qui reflètent la présence d'ensembles fortement cohésifs et faiblement couplés en utilisant des refactorisations. Intuitivement, les usages partagés d'attributs et d'invocations de méthodes sont un signe de cohésion alors que le couplage est directement exprimé par la dépendance d'une méthode sur une classe associée à travers ces méthodes et/ou attributs.

Dans le cas du Blob, nous appliquons les deux stratégies suivantes :

Premièrement, nous déplaçons les ensembles de méthodes cohésifs et disjoints et/ou les attributs qui peuvent être liés à une classe de données dans cette dernière. Les deux refactorisations suivantes décrivent de telles déplacements entre classes : *Déplacer une méthode* [Fowler, 1999, page 142] et *Déplacer un attribut* [Fowler, 1999, page 146].

Deuxièmement, nous organisons les sous-ensembles cohésifs qui ne sont pas liés aux classes de données dans des classes nouvellement créées. En plus des deux refactorisations précédentes, nous utilisons la refactorisation *Extraire une classe* [Fowler, 1999, page 149], qui consiste à créer une nouvelle classe et déplacer les attributs et les méthodes choisis de l'ancienne classe vers la nouvelle classe en utilisant les deux premières refactorisations.

Nous avons spécifié trois règles de refactorisation afin de construire les ensembles cohésifs incrémentaux en visitant le treillis de concepts. Ces règles sont appliquées en séquence : nous appliquons les deux premières règles qui traitent de l'accès des attributs par les méthodes en mode lecture/écriture et ensuite la règle qui considère les invocations de méthodes.

Règle 1. Les méthodes accédant en mode écriture au même ensemble d'attributs sont regroupées dans un unique ensemble cohésif.

Règle 2. Les méthodes accédant en mode lecture au même ensemble d'attributs sont regroupées dans un unique ensemble cohésif si le nombre d'attributs communs auxquels elles accèdent est plus grand que le nombre d'attributs auxquels elles accèdent séparément.

Ces deux règles sont inspirées de l'approche d'identification d'objets décrite dans [Sahraoui *et al.*, 1999] où le groupement des méthodes est basé sur les attributs accédés, par rapport au nombre d'attributs qu'elles accèdent séparément. Les ensembles cohésifs obtenus sont regroupés selon la règle suivante :

© 2008 Tous droits réservés.

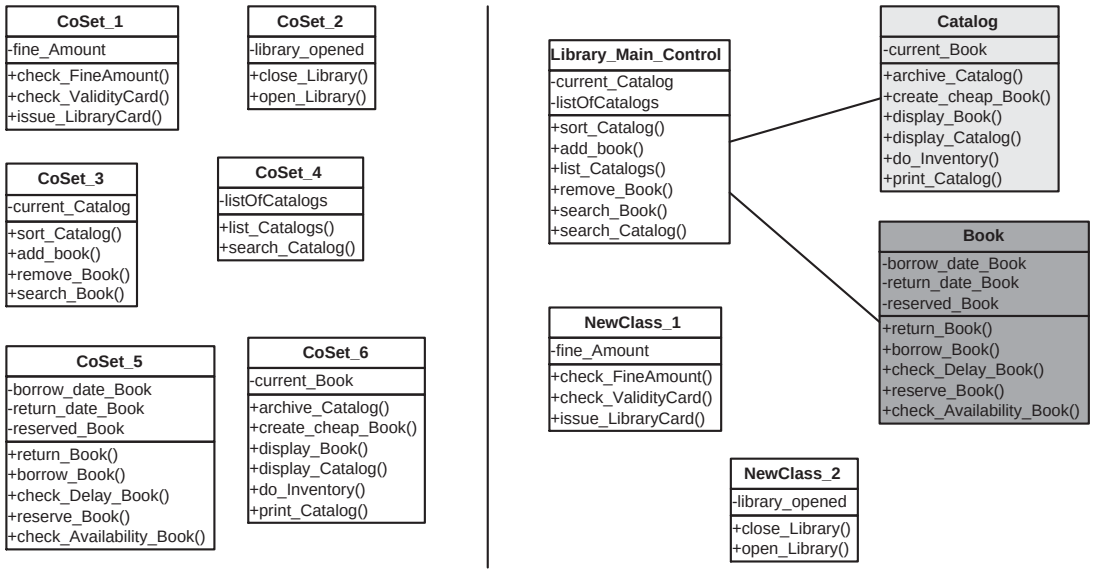
http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Règle 3. Les méthodes qui appellent le même ensemble de méthodes sont mises dans un ensemble cohésif unique si le nombre de méthodes appelées conjointement est supérieur au nombre de méthodes appelées séparément.

Par exemple, en appliquant les trois précédentes règles sur l'exemple illustratif de la bibliothèque, nous obtenons plusieurs ensembles cohésifs comme illustré sur la figure 4.8 à gauche. Les ensembles cohésifs qui doivent être migrés vers les classes de données sont montrés sur cette même figure à droite. Par manque de temps, cette dernière étape a été effectuée manuellement mais est prévue d'être automatisée dans les travaux futurs.

FIG. 4.8 – **Gauche :** Les ensembles cohésifs obtenus à partir de la classe `Library_Main_Control`. **Droite :** Déplacement de ces ensembles cohésifs vers les classes de données ou les nouvelles classes.



□

Nous détaillons ci-dessous l'implémentation de ces trois règles :

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Implémentation de la règle 1. Nous parcourons le treillis et enregistrons tous les concepts contenant des attributs avec le préfixe `W-`. Nous marquons tous ces concepts comme visités. Nous trions ensuite cette liste dans l'ordre décroissant du nombre de méthodes. Ainsi, les concepts contenant des méthodes qui accèdent en mode écriture à un grand nombre d'attributs sont traités en premier. Par exemple, le concept $c3$ dans la figure 4.7 est traité en premier à cause du concept lié $c13$ qui contient le plus grand nombre d'attributs accédés en écriture.

Pour chaque concept de la liste, nous créons un nouvel ensemble cohésif et appliquons la méthode `applyRuleWrite()`. Cette méthode déplace les attributs courants (`borrow_date_book()` et `return_date_book()` dans le concept $c13$) avec la refactorisation *Déplacer un attribut* et pour chaque méthode dans l'intension du concept courant (`borrow_Book()`) qui ne sont pas encore inclus dans un ensemble (c'est-à-dire pas encore visités), nous les déplaçons dans l'ensemble cohésif courant en utilisant la refactorisation *Déplacer une méthode*. Ensuite, récursivement, nous vérifions les parents du concept courant et éventuellement les enfants d'un parent. Les enfants d'un parent sont intéressants à explorer si le parent contient au moins un attribut `W-` également contenu dans le concept courant, par exemple, les enfants du parent $c13$ du concept $c3$. Enfin, nous réappliquons la méthode `applyRuleWrite()` sur les enfants.

Implémentation de la règle 2. Cette règle consiste à identifier des ensembles cohésifs de méthodes qui accèdent à un ensemble commun d'attributs en mode lecture. Pour chaque concept lié à des attributs communs en mode lecture et qui n'est pas encore visité c'est-à-dire pas encore traité lorsque nous appliquons la règle 1, et donc pas inclus dans un ensemble, nous calculons un ratio. Ce ratio permet de terminer si le nombre d'attributs communs accédés en lecture par un groupe de méthodes est plus grand que le nombre d'attributs auxquels elles accèdent séparément, comme décrit dans la règle 2. Il correspond au ratio du nombre d'attributs communs par le nombre total d'attributs. Nous calculons la moyenne de tous les ratios correspondant à chaque concept et retenons seulement les groupes de concepts qui ont une moyenne supérieure à 0,5. Ainsi, nous retenons seulement les concepts dont les méthodes accèdent à un nombre commun d'attributs supérieur en moyenne à leur propre nombre d'attributs. Nous obtenons ainsi une liste d'ensembles de concepts candidats que nous trions en ordre décroissant afin de traiter d'abord les concepts avec un ratio élevé. Pour chaque ensemble de concepts, nous créons un nouvel ensemble cohésif en déplaçant les méthodes et attributs en utilisant respectivement les refactorisations appropriées *Déplacer un attribut* and *Déplacer une méthode*.

Implémentation de la règle 3. Cette règle est similaire à la règle 2. La différence est que nous identifions des méthodes communes appelées par une ou plusieurs méthodes des ensembles cohésifs résultants construits à partir des règles 1 et 2. Nous calculons également un ratio et sélectionnons les meilleurs candidats et ensuite fusionnons les ensembles cohésifs selon la valeur de leur ratio.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

4.5 Expérimentations

Dans cette section, nous validons la technique COREX proposée pour la correction des défauts de conception de type Blob, et ainsi indirectement la méthode DECOR, sur quatre systèmes libres.

4.5.1 Hypothèse des expérimentations

L'hypothèse qui a été évaluée au cours de ces expérimentations est la suivante : *la précision des refactorisations suggérées par* COREX *est supérieure à 50%.* Cette hypothèse semble raisonnable étant donné l'aspect précurseur de ce type d'étude empirique.

4.5.2 Sujets des expérimentations

Ces expérimentations portent sur le Blob, défaut bien connu et commun caractérisé par une faible cohésion et un fort couplage. Néanmoins, ces expérimentations peuvent également s'appliquer à une dizaine d'autre défauts dont certains ont été listés dans la section 4.3.2.

4.5.3 Processus des expérimentations

Nous utilisons PADL [Guéhéneuc et Albin-Amiot, 2004] pour modéliser les classes du système à analyser et GALICIA v.2.1 [2005], pour construire et visualiser les treillis de concepts. GALICIA est une plate-forme libre en JAVA intégrant plusieurs outils pour créer, visualiser et mémoriser des treillis de concepts. Les deux outils communiquent au moyen de fichiers XML décrivant les données et les résultats. Un module supplémentaire à PTIDEJ [Guéhéneuc, 2005] génère des contextes dans le format XML de GALICIA, lesquels sont ensuite transformés en treillis, puis un module au sein de GALICIA permet d'exécuter les 3 règles et de renvoyer le résultat de la correction dans un fichier XML, où une restructuration du système est suggérée incluant les classes initiales et les classes nouvellement créées ainsi que leurs méthodes et leurs attributs associés.

4.5.4 Objets des expérimentations

Nous considérons quatre systèmes libres différents. Nous utilisons des systèmes disponibles gratuitement pour faciliter la comparaison et la réplication de nos expérimentations. Nous fournissons des informations sur ces systèmes dans le tableau 4.7.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Tableau 4.7 – Liste des systèmes.

Nom	Version	Lignes de code	Nombre de classes	Nombre d'interfaces
AZUREUS	2.3.0.6	191 963	1 449	546
Un client pair-à-pair implémentant le protocol BitTorrent				
LOG4J	1.2.1	10 224	189	14
Un framework extensible pour logger et débugger les applications Java				
LUCENE	1.4	10 614	154	14
Un moteur de recherche textuelle en JAVA				
NUTCH	0.7.1	19 123	207	40
Un moteur de recherche Web basé sur LUCENE				

□

4.5.5 Résultats des expérimentations

Dans AZUREUS, nous avons trouvé 41 Blobs en appliquant nos algorithmes de détection (*cf.* Tableau 3.4. page 70). Les classes sous-jacentes à ces Blobs ont un grand nombre d'attributs et de méthodes, sont faiblement cohésives et fortement couplées. Par exemple, la classe `DHTTransportUDPImpl` dans le paquetage `com.aelitis.azureus.core.dht.transport.udp.impl`, laquelle implémente une table de hachage distribuée (`DHT`) pour stocker de l'information de contact des pairs sur UDP, a une taille typiquement importante. Elle déclare 42 attributs et 66 méthodes pour 2 049 lignes de code. Elle a une cohésion moyenne de 0,542 et un fort couplage de 81 ($8^{ème}$ valeur la plus élevée parmi les 1 626 classes). Les classes de données qui entourent la large classe sont : `Average`, `HashWrapper` dans le paquetage `org.gudy.azureus2.core3.util` et `IpFilterManagerFactory` dans le paquetage `org.gudy.azureus2.core3.ipfilter`.

Le tableau 4.8 fournit les résultats de l'application de nos règles sur trois classes Blobs différentes détectées dans AZUREUS et sur deux classes Blobs détectées dans trois autres systèmes. Il est important de remarquer que les résultats fournis par notre technique ont été évalués manuellement : parmi les ensembles cohésifs en sortie, nous identifions ceux dont la sémantique pourrait être clairement établie afin de confirmer leur cohésion. Pour mesurer la précision des résultats de notre technique de correction, nous avons utilisé le ratio entre le nombre des réels ensembles cohésifs et le nombre total d'ensembles en sortie de la technique. Comme le tableau 4.8 l'indique, la précision peut varier largement entre 30 et 70% de suggestions correctes d'ensembles cohésifs. Néanmoins, la précision moyenne par système est supérieure à 50%, ce qui confirme notre hypothèse. Les ensembles cohésifs suggérés par notre technique incluent un nombre important de petits ensembles cohésifs, qui contiennent généralement au plus un attribut et une ou deux méthodes. Ceci explique pourquoi nous avons obtenu une faible précision de 31% pour un cas. Les autres ensembles cohésifs regroupent entre 10 et 20 attributs et/ou méthodes

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

et sont de bons candidats pour la création de nouvelles classes car ils définissent une responsabilité ou sémantique spécifique.

Tableau 4.8 – Précision des ensemble cohésifs extraits à partir de classes Blob dans quatre systèmes différents.

Système	Blob	Taille (nombre d'attributs et de méthodes)	Lignes de code	Nombre d'attributs et de méthodes déplacés	Nombre d'ensembles cohésifs	Nombre de réels ensembles cohésifs	Précision	Précision moyenne
Azureus v2.3.0.6	DHTTransportUDPImpl	(42+66) 108	2 049	(27+32) 59	10	7	70%	53%
	DHTControlImpl	(47+80) 127	1 868	(35+62) 97	19	11	58%	
	TRTrackerBTAnnouncerImpl	(36+47) 83	1 393	(24+33) 57	16	5	31%	
Log4j v1.2.1	LogBrokerMonitor	(29+105) 134	1 591	(23+85) 108	31	17	55%	52.5%
	Category	(9+53) 62	1 042	(8+44) 52	18	9	50%	
Lucene v1.4	IndexReader	(7+52) 59	593	(5+30) 35	4	2	50%	63.5%
	QueryParser	(36+48) 84	1 085	(24+37) 61	13	10	77%	
Nutch v0.7.1	FSNamesystem	(24+35) 59	1 211	(17+25) 42	18	9	50%	61.5%
	JobTracker	(22+31) 53	910	(17+18) 35	11	8	73%	

□

4.5.6 Menaces à la validité

Validité interne. Cette étude porte uniquement sur les défauts qu'il est possible de corriger avec des refactorisations. Cette limitation est liée à l'utilisation de l'analyse relationnelle de concepts. Dans des travaux futurs, nous envisageons d'explorer d'autres techniques pour la correction des défauts.

Validité externe. Tout comme dans la détection, une menace à la validité externe est l'utilisation exclusive de systèmes JAVA libres. Ce type de systèmes développés librement peuvent affecter les choix de conception et donc la présence de défauts. Néanmoins, le choix des systèmes libres est motivé par notre volonté de permettre la comparaison et la réplication de nos expérimentations.

Validité de construction. L'évaluation de la pertinence des refactorisations suggérées par la technique de correction est relativement subjective car elle peut différer d'un développeur à un autre. Cependant, seule une validation manuelle est requise afin d'évaluer

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

ce type d'expérimentations. Une solution à envisager pour évaluer plus objectivement les résultats est de faire valider les résultats par plusieurs développeurs indépendants.

Validité de conclusion. Les résultats et conclusions de cette étude semblent raisonnables dans le cadre d'une étude empirique préliminaire.

4.5.7 Discussion

Afin d'améliorer la robustesse de notre technique, nous avons besoin de définir des règles additionnelles liées à l'accès des attributs et l'invocation des méthodes par d'autres méthodes non seulement au sein d'une même classe mais également impliquant les autres classes associées. Par ailleurs, notre technique COREX pour la correction traite les entités des systèmes analysés de manière purement statique. Ainsi, nous avons besoin d'améliorer notre technique avec une analyse dynamique pour préserver le comportement d'un système. Finalement, la restructuration devrait être semi-supervisée par un expert car seuls les experts peuvent évaluer la pertinence de regrouper des éléments. La technique doit être vue comme un support pour restructurer un nombre important de données. Ainsi, nous partageons l'opinion de Snelting et Tip [2000] qu'une restructuration interactive réalisée par un ingénieur logiciel est plus appropriée.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Bilan

NOUS avons proposé une technique, nommée COREX, qui suit les étapes de la méthode DECOR et utilise l'ARC pour suggérer des restructurations afin de corriger certains défauts en utilisant des refactorisations. En particulier, nous avons montré comment notre technique peut aider à refactoriser des systèmes avec des défauts de conception de type Blob. L'ARC nous a permis de considérer dans les concepts d'un même treillis plusieurs aspects liés à la restructuration des défauts tels que l'accès des attributs par les méthodes et les invocations de méthodes. Nous avons présenté notre technique en l'illustrant avec un exemple d'un système de gestion d'une bibliothèque et nous l'avons validé sur AZUREUS v2.3.0.6 et trois autres systèmes. Nous avons montré qu'en utilisant l'ARC, notre technique pourrait suggérer des restructurations pertinentes pour améliorer la structure du système. La généralisation de nos résultats à d'autres défauts est discutée brièvement et sera développée dans des travaux futurs. Les travaux futurs incluront également d'évaluer plus de systèmes via notre approche et discuter les restructurations proposées avec leurs développeurs avant de les appliquer. Nous planifions également de réaliser des études quantitatives sur des compromis entre la cohésion et le couplage. Notre technique de correction COREX est originale car elle traite pour la première fois du problème de la correction des défauts de façon quasi-automatique.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Troisième partie

Conclusion et perspectives

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Chapitre 5

Conclusion

LES systèmes logiciels ne cessent d'être de plus en plus grands et de plus en plus complexes. De plus, ils évoluent de manière constante lors de la phase de maintenance. Cette évolution s'explique par la prise en compte des exigences des utilisateurs et l'ajout et/ou la modification de fonctionnalités. Cependant, la maintenance et l'évolution sont des activités coûteuses en temps et en ressources [Hanna, 1993 ; Pressman, 2001] et la présence de défauts rend ces activités encore plus difficiles. Ainsi, afin de faciliter ces activités et réduire les coûts de maintenance, il est important de détecter et corriger les défauts le plus tôt dans le cycle de développement. Des techniques ont été proposées pour la détection et la correction des défauts. Les techniques de détection consistent principalement à définir des règles de détection à appliquer sur le code source du système analysé. Les techniques de correction consistent à appliquer des refactorisations qui permettent de restructurer le code source du système analysé afin de corriger les défauts détectés. Or, les techniques de correction ne peuvent être appliquées directement suite à une détection des défauts car la phase qui consiste à identifier les restructurations n'a pas été étudiée et se fait manuellement par les ingénieurs logiciels. Ainsi, il existe un manque à combler entre la détection et la correction des défauts car celles-ci ont été traitées de façon isolée.

Nous avons relevé d'autres lacunes au sein des approches et outils proposés pour la détection et la correction des défauts. Tout d'abord, les approches existantes ne fournissent pas de langage de haut niveau pour spécifier les règles de détection des défauts selon le contexte du système analysé. Ensuite, la plupart des approches n'explicitent pas le passage des spécifications des défauts à leur détection ainsi que leur plate-forme de détection sous-jacente. Il est donc difficile de se comparer aux techniques existantes ou de les répliquer et les améliorer car ces techniques ne sont pas transparentes. Une autre lacune est qu'aucune approche ne tente d'automatiser la correction des défauts, ou plus précisément la suggestion des restructurations. Enfin, la validation de la détection a été réalisée de manière partielle sur peu de défauts et sur des systèmes propriétaires ; quant à la correction, aucune validation n'est présentée.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Pour répondre à ces lacunes, nous avons donc proposé, comme contribution principale, **DECOR**, une méthode qui définit, dans un même cadre, toutes les étapes nécessaires pour la détection et la correction des défauts.

Nous avons également fourni **des implémentations de référence de la méthode DECOR sous la forme d'une technique de détection, appelée DETEX, et d'une technique de correction, appelée COREX**.

La technique de détection DETEX est basée sur un langage spécifique au domaine résultant d'une analyse de domaine approfondie. C'est la première fois, dans la littérature, qu'un langage pour la détection des défauts s'appuie sur une telle analyse qui permet de capturer l'expertise du domaine lié aux défauts. Ainsi, grâce à ce langage, les ingénieurs logiciels peuvent spécifier les défauts en utilisant des abstractions de haut niveau tout en prenant en compte le contexte des systèmes analysés. De plus, en réponse à la lacune concernant le manque de transparence des techniques de détection, dans DETEX, nous avons détaillé la génération automatique des algorithmes de détection à partir des spécifications ainsi que la plate-forme de détection sous-jacente. Ce passage automatique des spécifications aux algorithmes de détection dispense de toute construction manuelle, coûteuse et peu réutilisable des algorithmes.

La technique de correction COREX permet de suggérer des restructurations pour corriger les défauts en utilisant des refactorisations. Cette technique est basée sur l'Analyse Relationnelle de Concepts (ARC), qui est une méthode algébrique pour identifier des groupes d'individus qui partagent des propriétés et des liens communs. Cette technique de correction est originale car elle traite pour la première fois dans la littérature du problème de la correction des défauts de manière quasi-automatique, et en particulier, la suggestion des restructurations pour corriger les défauts. De même, grâce à notre approche, il n'est plus nécessaire d'implémenter les règles de correction spécifiques à chaque défaut.

Ces deux techniques s'enchaînent et permettent d'une part de détecter les défauts et ensuite de suggérer des restructurations pour corriger les défauts détectés.

Une autre contribution concerne **la validation de la méthode DECOR en évaluant expérimentalement les techniques de détection DETEX et de correction COREX**. Dans DETEX, nous avons spécifié et détecté quatre défauts de conception (Blob, Functional Decomposition, Spaghetti Code, Swiss Army Knife) et leurs quinze défauts de code associés sur une dizaine de systèmes libres. Nous avons reporté pour le système XERCES v2.7.0 une précision moyenne de 60.5% et un rappel de 100% des algorithmes de détection et chacun des autres systèmes une précision supérieure à 50%. Nous avons ainsi montré que les algorithmes de détection sont raisonnablement efficaces et précis et permettent de détecter tous les défauts existants dans les systèmes analysés. C'est la première fois qu'une validation d'une telle ampleur est réalisée pour évaluer la détection des défauts. Nous avons également appliqué les algorithmes de détection sur ECLIPSE v3.1.2, démontrant leur passage à l'échelle et mettant en valeur le problème de balance entre les nombres

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

de classes suspectes, les précisions et le contexte de développement. Nous avons validé notre technique COREX sur des occurrences du défaut de conception Blob détectées dans quatre systèmes libres différents. Les résultats ont montré que COREX suggère une précision acceptable de restructurations pertinentes de l'ordre de 50%. Nous avons également discuté brièvement de la validité de la technique à une dizaine d'autres défauts que nous envisageons de valider empiriquement dans nos travaux futurs.

Une grande partie de notre travail, dans la détection des défauts, a porté sur la validation de la technique DETEX ainsi que sur le langage spécifique au domaine des défauts, des manques dans les travaux précédents. Nous maîtrisons ainsi bien le problème de la détection. Cependant, nous pensons qu'il serait fort utile que la communauté réunisse ses efforts pour construire un *benchmark*, tout comme dans la détection de code cloné, afin de comparer, tester et améliorer les approches existantes pour la détection des défauts.

En ce qui concerne la correction, nous avons répondu au besoin d'automatisation de la phase de suggestion des restructurations, mais la correction des défauts reste encore un problème difficile à maîtriser. En effet, bien que nous ayons validé la technique COREX sur des instances du défaut Blob et qu'elle soit applicable en théorie sur une dizaine d'autres défauts, nous n'avons pas démontré empiriquement sa validité sur d'autres défauts. Ainsi, il convient de considérer des techniques complémentaires à l'ARC afin de suggérer des restructurations pour un nombre plus large de défauts. Néanmoins, nos travaux sur la correction des défauts sont un premier effort vers la mise en lumière d'un problème au sein de la communauté et la nécessité d'y trouver une solution.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Chapitre 6

Perspectives

En termes de perspectives de recherche, nous envisageons, à court terme, d'améliorer notre méthode DECOR et de raffiner nos techniques de détection DETEX et de correction COREX afin d'obtenir une meilleure précision. Ensuite, à moyen terme, nous planifions un certain nombre d'extensions pour la détection des défauts. Enfin, pour le long terme, nous suggérons quelques pistes exploratoires.

6.1 Améliorations de la méthode DECOR et des techniques DETEX et COREX

Tout d'abord, nous envisageons d'appliquer et d'adapter DETEX et COREX à d'autres types d'entités que des modèles du code source tels que les diagrammes de classes. Ensuite, nous comptons fournir un classement des pires défauts détectés et des meilleures suggestions de restructurations, afin d'aider les ingénieurs logiciels à récolter et évaluer plus facilement et efficacement les résultats de la détection et de la correction. Nous envisageons également d'évaluer la méthode DECOR sur plus de défauts et de systèmes et de calculer la précision des techniques de détection DETEX et de correction COREX ainsi que le rappel sur d'autres systèmes que XERCES pour la détection.

Dans DETEX, nous proposons trois types d'améliorations. Tout d'abord, nous comptons fournir une description plus précise des propriétés utilisées dans les spécifications des défauts. En effet, nous voulons utiliser WORDNET, une base de données lexicale en anglais, pour élargir la liste des mots-clefs des propriétés lexicales et des grammaires de graphes pour préciser les propriétés structurelles. Ensuite, nous envisageons de concevoir une ontologie des défauts afin de bien organiser les concepts-clefs et leurs relations. Ainsi, nous visons à fournir un modèle de concepts-clefs bien structuré dans le domaine des défauts. Enfin, nous projetons d'améliorer la qualité et la performance du code source généré pour les algorithmes de détection en offrant plus de flexibilité dans la génération de code.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Dans COREX, nous comptons parfaire notre technique de trois manières. Tout d'abord, nous planifions de définir des règles additionnelles liées à l'accès des attributs et l'invocation des méthodes non seulement au sein d'une même classe mais également impliquant les autres classes associées. Ensuite, il nous reste également à automatiser l'étape qui consiste à migrer les ensembles cohésifs ou les classes nouvellement créées vers les classes de données pour la correction du Blob. Enfin, il est important de prendre en compte des critères de qualité autres que le couplage et la cohésion tels que la complexité.

6.2 Extensions

Nous prévoyons d'utiliser la détection des défauts pour étudier l'évolution et la qualité des systèmes, et pour fournir un cadre de comparaison pour les techniques de détection.

Nous avons étudié récemment l'évolution des défauts à travers des versions successives de différents systèmes. Nous avons observé que la hausse ou la baisse du nombre de défauts au cours des versions peut s'expliquer par des facteurs externes reportés dans les rapports de bogues tels que le passage à une version améliorée, l'ajout ou la modification de fonctionnalités et la correction de bogues. Nous comptons poursuivre cette étude car elle semble fort prometteuse pour étudier l'impact des défauts sur l'évolution des systèmes. De la même façon, nous envisageons de faire le même type d'études empiriques pour évaluer l'impact des défauts sur les coûts de maintenance.

Nous envisageons d'évaluer l'impact des défauts sur la qualité logicielle mais également évaluer si les défauts corrigés améliorent effectivement la qualité des systèmes. Nous avons déjà une expertise dans ce domaine car nous avons réalisé une étude empirique pour évaluer l'impact des patrons de conception sur la qualité logicielle [Foutse Khomh et Guéhéneuc, 2008 ; Khomh *et al.*, 2008]. Au cours de cette étude, il a été montré que contrairement à l'opinion générale, la mise en pratique des patrons de conception a un impact négatif sur certains attributs de qualité tels que la simplicité, la facilité d'apprentissage et la compréhension. Nous planifions donc de mener le même type d'étude empirique sur les défauts.

Nous envisageons également de concevoir un modèle de qualité qui tient compte de la présence de défauts pour évaluer la qualité des systèmes. Récemment, nous avons soumis un article à la revue *L'Objet* [Khomh *et al.*, 2008] qui propose une méthode, appelée DEQUALITE, pour construire des modèles de qualité qui permettent de mesurer la qualité des systèmes en prenant en compte non seulement les attributs internes des systèmes mais aussi leur conception. Cette méthode s'appuie sur une étude des patrons de conception pour prendre en compte la qualité de conception des systèmes. De la même façon, nous voulons évaluer la conception des systèmes en prenant en compte la présence des défauts.

Enfin, nous projetons de fournir un cadre de comparaison qui définit des critères pour classifier les approches de détection. Nous conduisons actuellement une étude sur les

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

outils de détection des défauts incluant plusieurs outils tels que REVJAVA, FINDBUGS, PMD, HAMMURAPI et LINT4J pour évaluer notre technique de détection, et indirectement notre méthode DECOR, par rapport aux outils existants. Une première comparaison est disponible dans l'état de l'art sur la détection.

6.3 Pistes exploratoires

Nous suggérons différentes pistes exploratoires incluant l'étude de nouvelles techniques et l'intégration à de nouveaux environnements afin de détecter et corriger les défauts.

Tout d'abord, nous proposons de réaliser des analyses dynamiques liées à l'exécution de code afin de prendre en compte les défauts liés au comportement des systèmes tels que l'anti-patron Poltergeist [Brown *et al.*, 1998, page 103]. Des travaux en cours sur les patrons de conception [Ng et Guéhéneuc, 2007] nous permettront de spécifier, détecter et corriger cette autre catégorie de défauts. Ainsi, nous envisageons d'étudier plus en détail les techniques utilisées dans l'identification des patrons de conception pour détecter ce type de défauts comportementaux.

Ensuite, nous prévoyons d'appliquer concrètement les restructurations en utilisant des environnements de transformation de programme comme KERMETA [Muller *et al.*, 2005], SPOON [Pawlak *et al.*, 2006] ou l'API d'ECLIPSE afin de s'assurer que ces restructurations sont réellement applicables et envisageables. Suite à ces restructurations, une analyse dynamique en plus de l'analyse statique peut être envisagée afin de s'assurer que le comportement du système n'a pas été modifié.

Enfin, nous avons commencé à examiner d'autres techniques pour la suggestion des restructurations telles que la programmation logique inductive (PLI) [Lavrač et Džeroski, 1994]. La PLI est une approche de l'apprentissage machine qui utilise les techniques de la programmation logique. Grâce à cette technique, il est possible, à partir d'une base de faits et d'exemples sur la manière de corriger un défaut, d'inférer de nouvelles règles pour corriger les défauts selon le contexte du système analysé.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Liste des publications

LES PUBLICATIONS qui ont trait à ce travail sont présentées dans la liste ci-dessous. Une liste complète et à jour de ces publications est également disponible à l'adresse suivante `http://www.naouelmoha.net/publications/index.html`.

Articles publiés dans des conférences internationales avec arbitrage

1. Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien. *A Domain Analysis to Specify Design Defects and Generate Detection Algorithms.* In José Fiadeiro and Paola Inverardi, editors. Proceedings of the 11^{th} International Conference on Fundamental Approaches to Software Engineering (FASE'08), pp. 276–291, March–April 2008, LNCS, Springer-Verlag.
2. Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. *Refactorings of Design Defects using Relational Concept Analysis.* In Raoul Medina and Sergei Obiedkov, editors. Proceedings of the 4^{th} International Conference on Formal Concept Analysis (ICFCA'08), February 2008, pp. 289–304, LNAI, Springer-Verlag.
3. Naouel Moha, Jihene Rezgui, Yann-Gaël Guéhéneuc, Petko Valtchev, and Ghizlane El Boussaidi. *Using FCA to Suggest Refactorings to Correct Design Defects.* Proceedings of the 4^{th} International Conference On Concept Lattices and Their Applications (CLA'06), In S. Ben Yahia and E. Mephu Nguifo, editors. October–November 2006, pp. 269–275, LNAI, Springer-Verlag.
4. Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. *Automatic Generation of Detection Algorithms for Design Defects.* Proceedings of the 21^{st} IEEE International Conference on Automated Software Engineering (ASE'06), In Sebastian Uchitel and Steve Easterbrook, editors. pp. 297–300, September 2006, IEEE Computer Society Press.

Articles publiés dans des conférences nationales avec arbitrage

1. Naouel Moha, Foutse Khomh, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. *Génération automatique d'algorithmes de détection des défauts de conception.* Actes du $14^{ième}$ colloque Langages et Modèles à Objets (LMO'08), In Mireille Blay-Fornarino, editors. Mars 2008, pp. 95–107, Éditions Cépaduès.
2. Naouel Moha, Duc-Loc Huynh, and Yann-Gaël Guéhéneuc. *Une taxonomie et un métamodèle pour la détection des défauts de conception.* Actes du $12^{ième}$ colloque Langages et Modèles à Objets (LMO'06), In Roger Rousseau, editors. Mars 2006, pp. 201–216, Hermès Science Publications.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Articles publiés dans des ateliers

1. Naouel Moha. *Detection and Correction of Design Defects in Object-Oriented Designs.* Doctoral Symposium, 21^{st} International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'08), October 2007.
2. Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. *Discussion on the Results of the Detection of Design Defects.* In Serge Demeyer, Yann-Gaël Guéhéneuc, Christian Lange, Kim Mens, Roel Wuyts, and Stéphane Ducasse, editors. Proceedings of the 8^{th} International ECOOP Workshop on Object-Oriented Reengineering (WOOR'07), July–August 2007, Springer-Verlag.
3. Naouel Moha. *Detection and Correction of Design Defects in Object-Oriented Architectures.* Doctoral Symposium, 20^{th} European Conference on Object-Oriented Programming (ECOOP'06), July 2006.
4. Naouel Moha and Saliha Bouden, and Yann-Gaël Guéhéneuc. *Correction of High-Level Design Defects with Refactorings.* Proceedings of the 7^{th} International ECOOP Workshop on Object-Oriented Reengineering (WOOR'06), July 2006, Springer-Verlag.
5. Naouel Moha, Duc-Loc Huynh, and Yann-Gaël Guéhéneuc. *A Taxonomy and a First Study of Design Pattern Defects.* Proceedings of the STEP International Workshop on Design Pattern Theory and Practice, September 2005.

Démonstrations d'outils

1. Naouel Moha and Yann-Gaël Guéhéneuc. *Ptidej and DECOR : Identification of Design Patterns and Design Defects.* 21^{st} International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07), October 2007.
2. Naouel Moha and Yann-Gaël Guéhéneuc. *Ptidej and DECOR : Identification of Design Patterns and Design Defects.* 22^{nd} International Conference on Automated Software Engineering (ASE'07), November 2007.
3. Naouel Moha and Yann-Gaël Guéhéneuc. *DECOR and Ptidej.* 16^{th} IBM Centers for Advanced Studies Conference (CASCON'06), October 2006.
4. Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, Duc-loc Huynh, Olivier Kaczor, Naouel Moha, and Samah Rached. *Ptidej : A Tool Suite.* 15^{th} IBM Centers for Advanced Studies Conference (CASCON'05), October 2005.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Quatrième partie

Annexes

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Annexe A

Répertoire des défauts

Alban TIBERGHIEN - Naouel MOHA - Tom MENS - Kim MENS

November 21th, 2007

Technical Report 1303, University of Montreal

Sommaire

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Afin de classer un certain nombre de défauts de conception, nous avons utilisé la classification proposée par Mäntylä [Mantyla, 2003] qui permet de mieux comprendre les différents défauts et les relations qui les lient. Nous avons décrit chaque défaut en suivant le patron suivant : le (ou les) nom(s) connu(s) de ce défaut, un identificateur (ID), une description, et dans certains cas des exemples.

A.1 The Bloaters

Cette famille de code smells caractérise des entités devenues si grande qu'elles sont devenues difficiles à gérer.

A.1.1 Long Method

Nom(s) : Long Method [Fowler, 1999, p. 76]
ID : `longMethod`

Description : Le principal problème d'une méthode trop longue est que le "fonctionnement" de celle-ci est difficile à définir. En effet, plus une méthode est longue, plus elle utilise de paramètres et de variables et il y a donc de fortes chances pour que cette méthode fasse plus de choses que son nom le suggère. Dans nos expériences, nous considérons qu'une méthode longue a plus de 35 instructions et de très longue une méthode avec plus de 90 instructions. Il est à noter qu'il s'agit de valeurs seuils et donc des valeurs abitraires : une méthode contenant 34 attributs peut également être considérée comme longue mais avec un degré de vérité de 95% par exemple[1]. Le fait d'éclater les méthodes trop longues en plus petites méthodes permet de mieux comprendre ce que le code fait. Cela permet donc une meilleure réutilisabilité et une meilleure flexibilité.

A.1.2 Large Class

Nom(s) : Large Class [Fowler, 1999, p. 78]
ID : `largeClass`

Description : Ces classes sont des classes ayant trop de responsabilités. Elles ont trop d'attributs de classe et/ou trop de méthodes. Le problème est que ces classes sont trop difficiles à maintenir et à comprendre à cause de leur taille. Nous considérons qu'une classe est large si elle a plus de 76 attributs et/ou méthodes et de très large une classe avec plus de 183 attributs et/ou méthodes. Il est à noter qu'il s'agit de valeurs seuils et donc des valeurs abitraires : une classe contenant 75 attributs peut également être considérée comme large mais avec un degré de vérité de 95% par exemple.

A.1.3 Primitive Obsession

Nom(s) : Primitive Obsession [Fowler, 1999, p. 81]

[1]Nous avons obtenu ce chiffre en analysant une dizaine de programmes. Nous avons mesuré le nombre d'instructions de toutes les méthodes de toutes les classes de chacun des programmes et nous avons appliqué des algorithmes de clustering pour extraire 3 clusters correspondant aux 3 catégories différentes de méthodes : les petites, les moyennes et les grandes. L'analyse de ces données nous a permis de récupérer les valeurs seuils associées à chacune des catégories.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

ID : `primitiveObsession`

Description : L'utilisation abusive des primitives tels que les chaînes de caractères, les réels, et les tableaux engendre un code difficile à modifier puisque très fortement lié à ces primitives. Il est possible de définir ces primitives sous forme de classes afin de fournir un niveau d'abstraction plus élévé pour rendre le code plus clair et plus simple.

A.1.4 Long Parameter List

Nom(s) : Long Parameter List [Fowler, 1999, p. 78]

ID : `longParameterList`

Description : Dans les langages de programmation procéduraux, l'utilisation de variables globales est considérée comme mauvaise. C'est pourquoi, dans ces langages, les fonctions ont souvent une longue liste de paramètres. L'approche objet a changé ce paradigme puisque les paramètres d'une méthode peuvent être réunis dans des objets. Ainsi, les données qu'une méthode a besoin sont directement récupérées à partir de l'objet passé en paramètre. Les listes de paramètres sont donc plus courtes et donc la méthode plus facile à utiliser et à comprendre. Nous considérons qu'à partir de 4 paramètres, il s'agit d'une longue liste de paramètres.

A.1.5 Data Clumps

Nom(s) : Data Clumps [Fowler, 1999, p. 81]

ID : `dataClumps`

Description : Il s'agit d'un ensemble de variables fortement cohésives. Ces variables se retrouvent constamment ensemble dans le code et peuvent être encapsulées dans des objets afin de réduire la taille des méthodes/classes.

Exemple(s) : les 3 entiers pour les couleurs RGB.

A.2 The Object-Oriented Abusers

Cette famille de code smells caractérise des cas où les choix d'implantation n'exploitent pas totalement les possibilités de la conception orientée objet.

A.2.1 Switch Statement

Nom(s) : Switch Statement [Fowler, 1999, p. 82]

ID : `switchStatement`

Description : L'utilisation de switch/case devrait être évitée dans la programmation objet. Souvent lorsqu'un code est pensé avec des switch/case, celui-ci est dupliqué partout dans la classe. Si un cas est ajouté/enlevé, il faut mettre à jour tous les switch/case. La notion de polymorphisme est une solution plus élégante pour traiter ce problème.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

A.2.2 Temporary Field

Nom(s) : Temporary Field [Fowler, 1999, p. 84]

ID : `temporaryField`

Description : Certains objets contiennent des attributs qui ne sont pas utilisés dans tous les cas. Ces attributs peuvent servir par exemple que pendant le déroulement d'un algorithme et le reste du temps, ils ne servent à rien et donc sont vides ou contiennent des données peu pertinentes. Généralement, tous les attributs doivent être utilisés. Si seule une partie des attributs est utilisée, le code est confus et difficile à comprendre. Ce défaut est une alternative aux listes de paramètres. Selon Mäntylä, ce défaut apparaît lorsqu'une variable est déclarée au niveau de la classe alors qu'elle devrait l'être au niveau de la méthode. Ceci viole le principe d'encapsulation.

A.2.3 Refused Bequest

Nom(s) : Refused Bequest [Fowler, 1999, p. 87]

ID : `refusedBequest`

Description : Ce code smell survient quand une sous-classe utilise peu ou pas du tout un attribut/méthode qu'elle a récupéré d'un parent par héritage. Typiquement, cela signifie que la hiérarchie des classes est fausse ou mal organisée.

A.2.4 Tradition Breaker

Nom(s) : Tradition Breaker [Lanza et Marinescu, 2006]

ID : traditionBreaker

Description : Ce défaut désigne une classe héritée qui brise la 'tradition' en fournissant un large ensemble de services qui n'ont pas de liens avec les services héritées par la classe mère.

A.2.5 Alternative Classes with Different Interfaces

Nom(s) : Alternative Classes with Different Interfaces [Fowler, 1999, p. 85]

ID : `alternativeClasses`

Description : Il s'agit de classes faisant la même chose mais ayant des interfaces différentes. Il s'agit d'une mauvaise utilisation de l'héritage.

A.2.6 Access to Children

Nom(s) : Access to Children

ID : `accessToChildren`

Description : Ce défaut désigne le cas où une classe utilise directement l'une de ses sous-classes. Ce défaut se présente quand une classe accède directement à une ou plusieurs des ses sous-classes. En principe ceci doit être évité car une classe n'est pas censée connaître ses sous-classes.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

A.2.7 Poor Usage of Abstract Classes

Nom(s) : Poor sage of Abstract Classes

ID : `poorUsageOfAbstractClasses`

Description : Ce défaut se produit lorsque des classes abstraites ne sont pas bien utilisées ou construites.

Exemple(s) :
- des classes abstraites sans méthodes abstraites ;
- des méthodes abstraites qui surchargent d'autres méthodes abstraites ;
- des classes abstraites sans aucune méthode ;
- des méthodes non-abstraites mais non implémentées.

A.2.8 Not Enough Information Hiding

Nom(s) : Not Enough Information Hiding

ID : `notEnoughInformationHiding`

Description : Ce défaut se produit quand une classe rend visible des méthodes ou des attributs alors qu'ils auraient dû être inaccessibles, de même, de manière plus générale, lorsqu'une classe expose des informations liées à l'interface ou à la base de données. L'utilisateur dispose donc d'information et de fonctionnalités qu'il ne devrait pas avoir.

Exemple(s) :
- l'accès à une structure de données pourrait être dangereux car l'utilisateur pourrait la manipuler comme il le souhaite et donc la corrompre ;
- les packages ou classes avec seulement (ou la plupart) des éléments publics sont susceptibles de ne pas assez encapsuler d'information.

A.2.9 Too Much Information Hiding

Nom(s) : Too Much Information Hiding

ID : `tooMuchInformationHiding`

Description : Ce défaut se produit quand une entité ne propose pas assez de fonctionnalités pour la manipuler correctement.

Exemple(s) : une classe dispose de trop de méthodes/attributs privées.

A.2.10 Poor Usage of Interfaces

Nom(s) : Poor Usage of Interfaces

ID : `poorUsageOfInterfaces`

Description : On observe ce défaut lorsque qu'une interface est mal utilisée.

Exemple(s) :
- une classe implémente une interface déjà implémentée par une autre classe ;
- une interface définit une méthode déjà déclarée dans une interface héritée.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

A.3 The Dispensables

Cette famille de code smells caractérise des entités superflues qui pourraient être supprimées du code source.

A.3.1 Lazy Class

Nom(s) : Lazy Class [Fowler, 1999, p. 83]
ID : `lazyClass`

Description : Ce sont des classes qui n'ont pas une grande utilité car elle ne contiennent peu ou pas de code fonctionnel. Les classes vides ou petites sont des cas spéciaux de ce type de défaut. Elles compliquent souvent la maintenance et la compréhension du projet. Dans ce cas, soit la classe est supprimée soit des fonctionnalités lui sont ajoutées.

A.3.2 Data Class

Nom(s) : Data Class [Fowler, 1999, p. 86]
ID : `dataClass`

Description : Ce sont des classes qui ne contiennent que des attributs et leurs accesseurs. Elles n'offrent aucune fonctionnalité particulière.

A.3.3 Duplicated Code

Nom(s) : Duplicated Code [Fowler, 1999, p. 76]
ID : `duplicatedCode`

Description : Ce défaut apparaît dans le code soit de manière explicite soit de manière subtile. La forme explicite correspond à du code redondant, alors que la forme subtile correspond à du code qui fait la même chose mais utilisant des combinaisons de données ou de comportements différents.

A.3.4 Dead Code

Nom(s) : Dead Code / Unused Code / Unnecessary Code
ID : `deadCode`

Description : Ce défaut désigne du code 'mort', inutilisé ou inutile. Il s'agit généralement de code qui n'est pas utilisé et dont personne n'ose supprimer car personne ne se souvient à quoi il sert.

A.3.5 Speculative Generality

Nom(s) : Speculative Generality [Fowler, 1999, p. 83] / Premature Generalization / Premature Abstraction
ID : `speculativeGenerality`

Description : Ce défaut existe quand on est en présence d'un code générique ou abstrait prévu pour des usages futures. Ce code pollue inutilement le système.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

A.4 The Change Preventers

Cette catégorie de défauts désigne les défauts ou structures de code qui rendent difficile la modification du logiciel.

A.4.1 Divergent Change

Nom(s) : Divergent Change [Fowler, 1999, p. 79]

ID : `divergentChange`

Description : Ce défaut désigne le fait qu'une classe donnée soit modifiée de différentes façons pour différentes raisons.

A.4.2 Shotgun Surgery

Nom(s) : Shotgun Surgery [Fowler, 1999, p. 80] / High Coupling

ID : `shotgunSurgery`

Description : Ce défaut désigne le fait que l'on soit obligé de modifier plusieurs classes lorsque l'on désire faire un changement donné.

A.4.3 Parallel Inheritance Hierarchies

Nom(s) : Parallel Inheritance Hierarchies [Fowler, 1999, p. 83]

ID : `parallelInheritanceHierarchies`

Description : Ce défaut est un cas spécial du Shotgun Surgery. Il désigne le fait que lorsqu'on crée une sous-classe d'une classe, on soit obligé de créer une sous-classe d'une autre classe.

A.5 The Couplers

Cette catégorie désigne les défauts qui sont fortement couplés.

A.5.1 Feature Envy

Nom(s) : Feature Envy [Fowler, 1999, p. 80]

ID : `featureEnvy`

Description : Ce défaut désigne le fait qu'une méthode fait beaucoup d'appels à d'autres classes afin d'obtenir des données et des fonctionnalités.

Exemple(s) :il peut s'agir d'une méthode qui invoque de nombreuses méthodes `get` d'une autre classe afin de pouvoir réaliser une opération.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

A.5.2 Inappropriate Intimacy

Nom(s) : Inappropriate Intimacy [Fowler, 1999, p. 85]
ID : `inappropriateIntimacy`

Description : Il s'agit de classes qui passent trop de temps à fouiller dans d'autres classes et donc entrent trop dans l'intimité de la classe. Plus spécifiquement, il s'agit de deux classes qui sont très fortement liées entre elles.

A.5.3 Message Chains

Nom(s) : Message Chains [Fowler, 1999, p. 84] / Law of Demeter
ID : `messageChains`

Description : Ce défaut désigne de longues séquences d'appels de méthodes ou de variables temporaires d'un objet à l'autre avant de pourvoir accéder à des données. Cette chaîne rend le code dépendant des relations entre un grand nombre d'objets potentiellement peu reliés.
Exemple(s) : chaînes d'invocation de méthodes tels que `A.getB().getC().do()`.

A.5.4 Middle Man

Nom(s) : Middle Man [Fowler, 1999, p. 85]
ID : `middleMan`

Description : Ce défaut désigne un abus du concept de délégation. Plus précisément, il s'agit d'une classe qui délègue trop de méthodes à une autre classe. La classe ne fait alors que passer des méthodes à d'autres classes.

A.5.5 Group of Interdependent Objects

Nom(s) : Group of Interdependent Objects
ID : `interdependentObjects`

Description : Ce défaut désigne une classe qui dépend immédiatement de beaucoup d'autres classes et d'autres classes dépendent immédiatement de celle-ci.

A.5.6 Circular Dependencies

Nom(s) : Circular Dependencies
ID : `circularDependencies`

Description : Une dépendance circulaire est lorsqu'on commence à partir d'une classe donnée A, on peut revenir sur cette même classe en suivant les liens de dépendance.

A.6 Others

Cette catégorie regroupe les défauts qui n'ont pas pu être classés dans l'une des catégories de cette classification.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

A.6.1 Incomplete Library Class

Nom(s) : Incomplete Library Class [Fowler, 1999, p. 86]

ID : `incompleteLibraryClass`

Description : Ce défaut désigne les responsabilités qui émergent du code et qui devraient être déplacées dans une classe de librairie. Mais, il semble difficile d'ajouter ces nouvelles responsabilités dans la classe de librairie.

A.6.2 Too Much Comments

Nom(s) : Too Much Comments

ID : `tooMuchComments`

Description : Dans certains cas, l'utilisation abusive de commentaires dans le code sont là pour expliquer du mauvais code.

A.6.3 Not Enough Comments

Nom(s) : Not Enough Comments

ID : `notEnoughComments`

Description : Le manque de commentaires dénote une négligence au niveau de la documentation du programme.

A.7 Anti-patrons

Les anti-patrons sont des défauts de conception plus complexes qui reposent sur l'existence de code smells.

A.7.1 The Blob

Nom(s) : Blob [Brown *et al.*, 1998, p. 73]

ID : `blob`

Description : Le Blob est une classe centralisant le traitement et donc par conséquent possède beaucoup d'attributs et de méthodes. En général, il y a peu de cohésion dans ces classes qui dépendent surtout d'autres classes où sont stockées les données. On remarque une absence de conception orientée objet (plutôt orientée procédurale). Ces classes sont difficiles à réutiliser et à tester. De plus, elles utilisent beaucoup de ressources. Cet anti-patron est souvent issu de l'évolution d'un prototype ou d'un développement incrémental d'un projet. Il est difficile de faire la différence entre l'utile et le superflu.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

A.7.2 Functional Decomposition

Nom(s) : Functional Decomposition [Brown *et al.*, 1998, p. 97] / Module Mimic

ID : `functionalDecomposition`

Description : On observe cet anti-patron lorsqu'un programmeur, à l'aise avec un langage procédural, transforme ces routines en classes ignorant ainsi la conception orientée objet. Le code est complexe et il n'y a aucune hiérarchie de classes. De plus, la classe ne propose qu'une seule fonctionnalité. On peut aussi remarquer que les attributs sont "private" et que les classes ont des noms de fonction tels que 'Calculat_Interest', 'Display_Table'. Le non-respect des principes de la programmation orientée objet rend le code difficile à maintenir, à documenter, à réutiliser et à tester.

A.7.3 Spaghetti Code

Nom(s) : Spaghetti Code [Brown *et al.*, 1998, p. 119]

ID : `spaghettiCode`

Description : C'est un système où on trouve peu de structure : peu/pas d'héritage, de réutilisation et/ou de polymorphisme. Le système inclut un petit nombre d'objets avec des méthodes trop grandes qui sont appelées une seule fois. Il y a un faible degré d'interaction entre les objets. Les méthodes n'ont pas de paramètres et utilisent des classes ou des variables globales pendant le traitement. Ces codes ne sont, en général, pas réutilisables. Le coût de maintenance de tels programmes peut être supérieur au coût de la réalisation d'une nouvelle solution.

A.7.4 Swiss Army Knife

Nom(s) : Swiss Army Knife [Brown *et al.*, 1998, p. 197]

ID : `swissArmyKnife`

Description : Il s'agit d'une classe pour laquelle le programmeur a essayé de fournir toutes les signatures de méthodes possibles dans le but de répondre à tous les besoins. Tel un couteau suisse, beaucoup de fonctionnalités sont disponibles mais peu sont utilisées concrètement. Cet outil prend donc de la place pour rien et en devient difficile à manipuler.

Exemple(s) : les classes utilitaires.

A.8 Code Level Defects

Les défauts au niveau code sont des problèmes à un très fin niveau de granularité qui sont localisés au niveau des lignes de code et qui peuvent découler d'une mauvaise façon d'utiliser des constructions de language de programmation.

A.8.1 Copying / Assignment / Initialisation

Nom(s) : Copying / Assignment / Initialisation

ID : `copyingAssignmentInitialisation`

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Description : Cette catégorie comprend tous les défauts au niveau code liés à un mauvais usage des initialisations, des affectations et des copies de variables. Il est difficile de donner une liste exhaustive des problèmes de cette catégorie car les différents outils de détection considèrent des types de problèmes de cette catégorie bien différents.

Exemple(s) :
- l'affectation d'une variable qui n'a pas d'effet car la variable n'est pas utilisée dans la suite du programme ou car elle est immédiatement assignée à une autre valeur ;
- l'utilisation de variables non initialisées ;
- l'affectation de variables à null ;
- la manipulation d'une copie de l'objet au lieu de l'objet lui-même (et inversement) ;
- une copie superficielle de l'objet a été utilisée au lieu d'une copie intégrale (et inversement) ;
- des champs immuables.
- ...

A.8.2 Equality and Identity Testing

Nom(s) : Equality and Identity Testing

ID : `equalityIdentityTesting`

Description : Cette catégorie comprend tous les défauts de niveau code qui sont liés à l'identité et l'égalité entre objets.

A.8.3 Problematic Conditionals

Nom(s) : Problematic Conditionals

ID : `problematicConditionals`

Description : Cette catégorie fait référence aux conditions logiques du type (if ...else ...) qui sont compliquées inutilement.

Exemple(s) :
- Instructions `if` imbriquées
- Instructions conditionnelles sous la forme négative (if not ...else ...)
- ...

A.8.4 Pathological Switch

Nom(s) : Pathological Switch

ID : `pathologicalSwitch`

Description : Cette catégorie est liée à la précédente, mais fait référence à une mauvaise utilisation des instructions `switch`.

Exemple(s) :
- instruction `switch` sans `default` ;
- instruction `default` vide ;
- ...

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Attention, ce défaut ne doit pas être confondu avec le défaut *Switch Statement* dans la catégorie *The Object-Oriented Abusers*.

A.8.5 Constuctor Problems

Nom(s) : Constuctor Problems

ID : `constuctorProblems`

Description : Cette catégorie fait référence aux défauts liés à un mauvais usage des constructeurs en Java.

Exemple(s) :
- passer les constructeurs à null ;
- trop de constructeurs.

A.8.6 Instantiation Problems

Nom(s) : Instantiation Problems

ID : `instantiationProblems`

Description : Cette catégorie fait référence aux problèmes liés à l'instanciation des classes, c.-à-d. la création des objets sous forme d'instances de classes.

Exemple(s) :
- des classes publiques avec aucun constructeur qui peut être instantié à l'extérieur ;
- des classes feuilles qui n'ont pas d'instances semblent ne pas être utilisées.

A.8.7 Memory Management Problems

Nom(s) : Memory Management Problems

ID : `memoryManagementProblems`

Description : Cette catégorie fait référence aux défauts liés à la gestion de la mémoire.

Exemple(s) : les problèmes liés à l'utilisation des `finalizer` en Java.

A.8.8 Exception Handling Problems

Nom(s) : Exception Handling Problems

ID : `exceptionHandlingProblems`

Description : Le mécanisme de gestion des exceptions de Java (`try`, `catch`, `throw`) est aussi une source commune de problèmes.

Exemple(s) :
- une instruction `return` dans un bloc `finally` ;
- une instruction `throw` dans un bloc `finally` ;
- des blocs `catch` vides ;

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

- des blocs `finally` vides ;
- des blocs au niveau des `catch` et des `throw` ;
- des pauvres propagations d'exception.
- ...

A.8.9 Concurrency Problems

Nom(s) : Concurrency Problems

ID : `concurrencyProblems`

Description : Cette catégorie regorupe les problèmes liés au mécanisme de concurrence utilisé en Java tels que les threads.

A.8.10 Scoping Problems

Nom(s) : Scoping Problems

ID : `scopingProblems`

Description : Cette catégorie décrit les problèmes liés à la portée des variables et des maéthodes au sein d'une classe.

Exemple(s) :

- des variables locales qui masquent les attributs de la classe ;
- les paramètres des méthodes qui ont les mêmes noms que les attributs de la class' ;
- des déclarations de champs qui masquent des champs qui sont accessibles dans un super type.

A.8.11 Unrespected Naming Rules

Nom(s) : Unrespected Naming Rules

ID : `namingRules`

Description : Cette catégorie regroupe tous les défauts liés aux règles de nommage.

Exemple(s) :

- des méthodes ou des classes avec de long noms ;
- une méthode qui a le même nom qu'un constructeur ;
- des noms d'identificateurs vagues ;
- des variables qui ont des noms identiques à ceux de types.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Bibliographie

NOUS avons choisi d'utiliser les langues des documents référencés pour les entrées de la bibliographie ; ainsi, une entrée référençant un document en français est en français, une entrée référençant un document en anglais est en anglais. Cependant, la référence à l'ouvrage est toujours donnée en français quelle que soit la langue du document référencé. Tous les sites Web listés dans les références ci-dessous ont été accédés le 23 juillet 2008.

[Albin-Amiot *et al.*, 2002] cité pages 53, 58
Hervé Albin-Amiot, Pierre Cointe et Yann-Gaël Guéhéneuc. Un méta-modèle pour coupler application et détection des design patterns. Michel Dao et Marianne Huchard, éditeurs, *Actes du 8^{e} colloque Langages et Modèles à Objets*, volume 8, numéro 1-2/2002 de *RSTI – L'objet*, pages 41–58. Hermès Science Publications, janvier 2002.

[Albin-Amiot et Guéhéneuc, 2001] cité page 19
Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Pim van den Broek, Pavel Hruby, Motoshi Saeki, Gerson Sunyé, and Bedir Tekinerdogan, editors, *Proceedings of the 1st ECOOP workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente, October 2001. TR-CTIT-01-35.

[Alikacem et Sahraoui, 2006a] cité pages 5, 17, 20, 27, 46
El Hachemi Alikacem et Houari Sahraoui. Détection d'anomalies utilisant un langage de description de règle de qualité. Roger Rousseau, Christelle Urtado et Sylvain Vauttier, éditeurs, *actes du 12^{e} colloque Langages, Modèles, Objets*, pages 185–200. Hermès Science Publications, mars 2006.

[Alikacem et Sahraoui, 2006b] cité page 17
El Hachemi Alikacem and Houari Sahraoui. Generic metric extraction framework. In *Proceedings of the 16th International Workshop on Software Measurement and Metrik Kongress (IWSM/MetriKon)*, pages 383–390, 2006.

[Allen et Garlan, 1997] cité page 20
Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[Analyst4j, 2008] cité page 19
Analyst4j, February 2008. `http://www.codeswat.com/`.

[Aqris, 2002] cité page 25
Aqris. Refactorit, 2002. `http://www.refactorit.com/`.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Arévalo, 2005] cité page 23
Gabriela Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. Ph.D. thesis, University of Berne, January 2005.

[Arnold, 1989] cité page 6
Robert S. Arnold. Software restructuring. In *Proceedings of the IEEE*, volume 77, pages 607–617. IEEE Computer Society Press, April 1989.

[Artho, 2004] cité pages 18, 19
Artho. Jlint, July 2004. `http://artho.com/jlint/`.

[Arthur, 1988] cité page 4
Lowell Jay Arthur. *Software Evolution: The Software Maintenance Challenge*. Wiley-Interscience, New York, NY, USA, 1988. ISBN: 0-471-62871-9.

[Barbut et Monjardet, 1970] cité page 73
M. Barbut and B. Monjardet. *Ordre et classification — Algèbre et combinatoire (2 tomes)*. Hachette, 1970.

[Bart Du Bois et Demeyer, 2004] cité pages 6, 82
Jan Verelst Bart Du Bois and Serge Demeyer. Refactoring - improving coupling and cohesion of existing code. In *Proceedings of the* 11^{th} *IEEE Working Conference on Reverse Engineering*, pages 144–151, 2004.

[Beyer *et al.*, 2005] cité page 19
Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *Transactions on Software Engineering*, 31(2):137–149. IEEE Computer Society Press, February 2005.

[Beyer *et al.*, 2007] cité page 19
Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. Journal on Software Tools for Technology Transfer*, 9:505–525, 2007. Invited to special issue of selected papers from FASE 2005.

[Birkhoff, 1940] cité page 73
Garret Birkhoff. *Lattice Theory*, volume 25 of *Colloquium publications*. American Mathematical Society, 1940.

[Boroday *et al.*, 2005] cité page 12
Sergiy Boroday, Alexandre Petrenko, Jagmit Singh, and Hesham Hallal. Dynamic analysis of Java applications for multithreaded antipatterns. In *Proceedings of the* 3^{rd} *International Workshop On Dynamic Analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press. ISBN: 1-59593-126-0.

[Brant, 1997] cité pages 18, 19
John Brant. Smalllint, April 1997. `http://st-www.cs.uiuc.edu/users/brant/Refactory/Lint.html`.

[Brown *et al.*, 1998] cité pages 4, 12, 16, 27, 36, 37, 39, 42, 60, 72, 83, 105, 118, 119
William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, March 1998. ISBN: 0-471-19713-0.

[Brown, 1996] cité page 19
Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical report TR-96-07, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1996.

[Bruno *et al.*, 2007] cité page 16
Giulia Bruno, Paolo Garza, Elisa Quintarelli, and Rosalba Rossato. Anomaly detection in xml

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

databases by means of association rules. In *DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications*, pages 387–391, Washington, DC, USA, 2007. IEEE Computer Society. ISBN: 0-7695-2932-1.

[Chambers *et al.*, 1983] cité page 45
John M. Chambers, William S. Clevelmd, Beat Kleiner, and Paul A. Tukey. *Graphical methods for data analysis*. Wadsworth International, 1983. ISBN: 0-534-98052-X.

[CheckStyle, 2004] cité page 19
CheckStyle, 2004. `http://checkstyle.sourceforge.net`.

[Chen et Wagner, 2002] cité page 19
Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, 2002.

[Chidamber et Kemerer, 1994] cité pages 44, 82
Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Los Alamitos, CA, USA, 1994. IEEE Computer Society.

[Consel et Marlet, 1998] cité page 47
Charles Consel and Renaud Marlet. Architecturing software using: A methodology for language development. *Lecture Notes in Computer Science*, 1490:170–194, September 1998.

[Cordy *et al.*, 2002] cité page 26
James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the txl transformation system. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[Dao *et al.*, 2004] cité pages 72, 77
Michel Dao, Marianne Huchard, Mohamed Rouane Hacene, Cyril Roume, and Petko Valtchev. Improving generalization level in uml models iterative cross generalization in practice. In *Proceedings of* 12^{th} *International Conference on Conceptual Structures (ICCS'04)*, pages 346–360. LNCS 3127, Springer, 2004.

[Dashofy *et al.*, 2005] cité page 20
Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):199–245, New York, NY, USA, 2005. ACM.

[DECOR, 2006] cité pages 48, 69
DECOR, September 2006. `http://www.naouelmoha.net/DECOR/DECOR.htm`.

[Detlefs, 1996] cité page 19
David L. Detlefs. An overview of the extended static checking system. In *Proceedings of the First Formal Methods in Software Practice Workshop (1996)*, 1996.

[Dhambri *et al.*, 2008] cité page 17
Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In *Proceedings of the* 12^{th} *European Conference on Software Maintenance and Reengineering, Tampere, Finland*, pages 279–283. IEEE Computer Society, April 2008.

[Ducasse *et al.*, 2000] cité page 24
Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *CoSET '00: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, June 2000.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Dudney *et al.*, 2003] cité page 12
Bill Dudney, Stephen Asbury, Joseph Krozak, and Kevin Wittkopf. *J2EE AntiPatterns*. Wiley, 2003. ISBN: 0-471-14615-3.

[Evans, 1996] cité pages 18, 19
David Evans. Static detection of dynamic memory errors. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 44–53, New York, NY, USA, 1996. ACM Press. ISBN: 0-89791-795-2.

[Fenton et Neil, 1999] cité page 4
Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *Software Engineering*, 25(5):675–689, 1999.

[Fenton et Pfleeger, 1997] cité page 82
Norman Fenton and Shari Lawrence Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997. ISBN: 0-534-95600-9.

[Florijn, 2002] cité pages 5, 19
Gert Florijn. Revjava – design critiques and architectural conformance checking for java software, 2002. White Paper. SERC, the Netherlands.

[Foster, 1993] cité page 4
John Foster. *Cost Factors in Software Maintenance*. Phd thesis, Computer Science Department, University of Durham, 1993.

[Fowler, 1999] cité pages 4, 6, 12, 13, 16, 17, 22, 24, 27, 36, 39, 60, 83, 89, 111, 112, 113, 115, 116, 117
Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999. ISBN: 0-201-48567-2.

[Frakes et Baeza-Yates, 1992] cité page 60
William B. Frakes and Ricardo A. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[Fraser *et al.*, 1992] cité page 56
Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, New York, NY, USA, 1992. ACM.

[Frederick P. Brooks, 1975] cité page 4
Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, Reading, MA , USA, 1975. ISBN: 0201835959.

[FXCop, 2006] cité page 19
FXCop, June 2006. `http://www.binarycoder.net/fxcop/index.html`.

[Galicia, 2005] cité page 93
Galicia. Galicia, September 2005. `http://sourceforge.net/projects/galicia/`.

[Gamma *et al.*, 1994] cité pages 4, 17, 41, 51
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994. ISBN: 0-201-63361-2.

[Ganter et Wille, 1997] cité pages 75, 76
Bernhard Ganter and Rudolf Wille. Applied lattice theory: Formal concept analysis, 1997.

[Ganter et Wille, 1999] cité pages 22, 72, 73, 74
Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Heidelberg, 1999. ISBN: 3540627715.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Garlan *et al.*, 1995] cité page 20
David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[Geraci, 1991] cité page 4
Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers Inc., The, 1991. ISBN: 1559370793.

[Godin et Mili, 1993] cité page 23
Robert Godin and Hafedh Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '93)*, pages 394–410, New York, NY, USA, 1993. ACM Press. ISBN: 0-89791-587-9.

[Godin et Valtchev, 2005] cité page 23
Robert Godin and Petko Valtchev. Formal concept analysis-based normal forms for class hierarchy design in oo software development. In R. Wille B. Ganter, G. Stumme, editor, *Formal Concept Analysis: Foundations and Applications*, chapter 16, pages 304–323. Springer Verlag, 2005.

[Grant et Cordy, 2003] cité page 26
Scott Grant and James R. Cordy. An interactive interface for refactoring using source transformation. In *Proceedings of the First International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE'03)*, pages 30–33, November 2003.

[Grotehen et Dittrich, 1997] cité page 25
Thomas Grotehen and Klaus R. Dittrich. The meTHOOD approach: Measures, transformation rules, and heuristicsfor object-oriented design. Technical report ifi-97.09, University of Zurich, 1997.

[Guéhéneuc *et al.*, 2004] cité page 53
Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.

[Guéhéneuc et Albin-Amiot, 2001] cité page 39
Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *Proceedings of the* 39^{th} *Conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

[Guéhéneuc et Albin-Amiot, 2004] cité pages 53, 67, 93
Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM Press, October 2004.

[Guéhéneuc et Antoniol, 2007] cité pages 19, 53
Yann-Gaël Guéhéneuc and Giuliano Antoniol. Demima: A multi-layered framework for design pattern identification. *Transactions on Software Engineering*. IEEE Computer Society Press, December 2007. *Accepted.*

[Guéhéneuc, 2005] cité pages 64, 93
Yann-Gaël Guéhéneuc. PTIDEJ: Promoting patterns with patterns. In Mohamed E. Fayad, editor, *Proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

[Halstead, 1977] cité page 4
Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN: 0444002057.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Hammurapi, 2007] cité page 19
Hammurapi, October 2007. http://www.hammurapi.biz/.

[Hanna, 1993] cité pages 4, 99
Mary Hanna. Maintenance burden begging for a remedy. *Datamation*, pages 53–63, April 1993.

[Hedin, 1997] cité page 19
Görel Hedin. Language support for design patterns using attribute extension. In Jan Bosch and Stuart Mitchell, editors, *Proceedings of the 1st ECOOP workshop on Language Support for Design Patterns and Frameworks*, pages 137–140. Springer-Verlag, June 1997.

[Holt, 1998] cité page 17
Richard C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, pages 210–219, Honolulu, HI, USA, October 1998. IEEE Computer Society. ISBN: 0-8186-8967-6.

[Hovemeyer et Pugh, 2004] cité pages 5, 19
David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, New York, NY, USA, 2004. ACM Press.

[Huchard *et al.*, 2007] cité pages 72, 73, 77
Marianne Huchard, Cyril Roume, Amine Rouane Hacène, and Petko Valtchev. Relational concept discovery in structured datasets. *Annals of Mathematics and Artificial Intelligence*, 49(1-4):39–76, 2007.

[Huchard et Leblanc, 2000] cité page 23
Marianne Huchard and Hervé Leblanc. Computing interfaces in java. In *ASE*, pages 317–320, 2000.

[Instantiations, Inc., 2005] cité page 19
Instantiations, Inc. Codepro analytix, 2005. http://www.instantiations.com/codepro/analytix/about.html.

[Jackson, 1995] cité page 19
Daniel Jackson. Aspect: detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, New York, NY, USA, 1995. ACM Press.

[Jahnke et Zündorf, 1997] cité page 19
Jens H. Jahnke and Albert Zündorf. Rewriting poor design patterns by good design patterns. In Serge Demeyer and Harald C. Gall, editors, *Proceedings the 1st ESEC/FSE workshop on Object-Oriented Reengineering*. Distributed Systems Group, Technical University of Vienna, September 1997. TUV-1841-97-10.

[Johnson, 1977] cité page 18
Stephen C. Johnson. Lint, a C program checker. Technical report Computing Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, USA, December 1977.

[Jorwekar *et al.*, 2007] cité page 16
Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1263–1274. VLDB Endowment, 2007. ISBN: 978-1-59593-649-3.

[JRefactory, 2000] cité pages 25, 28
JRefactory, October 2000. http://sourceforge.net/projects/jrefactory.

[Jussien et Barichard, 2000] cité page 19
Narendra Jussien and Vincent Barichard. The PaLM system: Explanation-based constraint programming. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS:*

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Techniques foR Implementing Constraint Programming Systems, pages 118–133. School of Computing, National University of Singapore, Singapore, September 2000. TRA9/00.

[Keller *et al.*, 1999] cité page 19
Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In David Garlan and Jeff Kramer, editors, *Proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.

[Kent, 2002] cité page 34
Stuart Kent. Model driven engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[Khomh *et al.*, 2008] cité page 104
Foutse Khomh, Naouel Moha et Yann-Gaël Guéhéneuc. Dequalite : Mthode de construction de modles de qualit prenant en compte la conception des systmes. *Revue des Sciences et Technologies de l'Information (RSTI) – L'Objet*. Herms, Lavoisier, 2008. Subtmitted.

[Kiczales *et al.*, 1991] cité pages 34, 50
Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1^{st} edition, July 1991. ISBN: 0-7923-8067-3.

[Kirk *et al.*, 2006] cité pages 23, 28
Douglas Kirk, Marc Roper, and Neil Walkinshaw. Using attribute slicing to refactor large classes. In David W. Binkley, Mark Harman, and Jens Krinke, editors, *Beyond Program Slicing*, number 05451 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[Klimas *et al.*, 1996] cité page 4
Edward J. Klimas, Suzanne Skublics, and David A. Thomas. *Smalltalk with Style*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN: 0-13-165549-3.

[Klocwork, 2006] cité page 19
Klocwork. Klocwork insight, 2006. `http://www.klocwork.com`.

[Kullbach et Winter, 1999] cité page 19
Bernt Kullbach and Andreas Winter. Querying as an enabling technology in software reengineering. In Paolo Nesi and Chris Verhoef, editors, *Proceedings of the* 3^{rd} *Conference on Software Maintenance and Reengineering*, pages 42–50. IEEE Computer Society Press, March 1999.

[Langelier *et al.*, 2005] cité page 17
Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In Tom Ellman and Andrea Zisma, editors, *Proceedings of the* 20^{th} *International Conference on Automated Software Engineering*. ACM Press, November 2005.

[Lanza et Marinescu, 2006] cité pages 17, 41, 113
Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. ISBN: 3-540-24429-8.

[Lavrač et Džeroski, 1994] cité page 105
Nada Lavrač and Saso Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, NY, 10001, 1994.

[Lehman, 1996] cité page 4
Meir M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Lientz et Swanson, 1980] cité page 4
Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. ISBN: 0201042053.

[Man Machine Systems, 2005] cité page 19
Man Machine Systems. Jstyle, 2005. `http://www.mmsindia.com/jstyle.html`.

[Mantyla, 2003] cité pages 39, 111
Mika Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study.* Ph.D. thesis, Helsinki University of Technology, 2003.

[Marinescu, 2002] cité page 46
Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Politehnica University of Timisoara, October 2002.

[Marinescu, 2004] cité pages 5, 17, 27
Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society Press, 2004.

[McCabe, 1976] cité page 12
Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[Mens *et al.*, 2006] cité page 19
Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views – a case study. In Robert S. Ledley, editor, *Computer Languages, Systems, and Structures*. Elsevier, June 2006.

[Moha *et al.*, 2006a] cité page 52
Naouel Moha, Yann-Gaël Guéhéneuc, and Pierre Leduc. Automatic generation of detection algorithms for design defects. In Sebastian Uchitel and Steve Easterbrook, editors, *Proceedings of the* 21^{st} *Conference on Automated Software Engineering*. IEEE Computer Society Press, September 2006. Short paper.

[Moha *et al.*, 2006b] cité pages 50, 52
Naouel Moha, Duc-Loc Huynh et Yann-Gaël Guéhéneuc. Une taxonomie et un métamodèle pour la détection des défauts de conception. Roger Rousseau, éditeur, *actes du* 12^{e} *colloque Langages et Modèles à Objets*, pages 201–216. Hermès Science Publications, March 2006.

[Moha *et al.*, 2008a] cité pages 36, 52
Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Francoise Le Meur, and Laurence Duchien. A domain analysis to specify design defects and generate detection algorithms. In Jos Fiadeiro and Paola Inverardi, editors, *Proceedings of the* 11^{th} *International Conference on Fundamental Approaches to Software Engineering*, pages 276–291. Springer-Verlag, 2008.

[Moha *et al.*, 2008b] cité page 52
Naouel Moha, Foutse Khomh, Yann-Gaël Guéhéneuc, Laurence Duchien et Anne-Françoise Le Meur. Génération automatique d'algorithmes de détection des défauts de conception. Mireille Blay-Fornarino, éditeur, *Actes du* 14^{e} *colloque Langages et Modèles à Objets*, pages 95–107. Éditions Cépaduès, mars 2008.

[Muller *et al.*, 2000] cité page 59
Hausi A. Muller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne D. Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE — Future of SE Track*, pages 47–60, 2000.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Muller *et al.*, 2005] cité page 105
Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jzquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *MoDELS/UML'2005*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, 2005. Springer.

[Munro, 2005] cité pages 5, 17, 27
Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In Filippo Lanubile and Carolyn Seaman, editors, *Proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[Murphy et Notkin, 1996] cité page 53
Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, New York, NY, USA, 1996. ACM.

[Ng et Guéhéneuc, 2007] cité pages 42, 105
Janice Ka-Yee Ng and Yann-Gaël Guéhéneuc. Identification of behavioral and creational design patterns through dynamic analysis. In Andy Zaidman, Abdelwahab Hamou-Lhadj, and Orla Greevy, editors, *Proceedings of the* 3^{rd} *International Workshop on Program Comprehension through Dynamic Analysis*, pages 34–42. Delft University of Technology, October 2007. TUD-SERG-2007-022.

[Niere *et al.*, 2002] cité page 19
Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In Michal Young and Jeff Magee, editors, *Proceedings of the* 24^{th} *International Conference on Software Engineering*, pages 338–348. ACM Press, May 2002.

[Opdyke, 1992] cité page 24
William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[Patcha et Park, 2007] cité page 16
Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw.*, 51(12):3448–3470, New York, NY, USA, 2007. Elsevier North-Holland, Inc.

[Pawlak *et al.*, 2006] cité page 105
Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program analysis and transformation in java. Technical report Technical Report 5901, INRIA, May 2006.

[Perry et Wolf, 1992] cité page 4
Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. In Peter G. Neumann, editor, *Software Engineering Notes*, 17(4):40–52. ACM Press, October 1992.

[Philippow *et al.*, 2005] cité page 19
Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software and System Modeling*, 4(1):55–70. Springer-Verlag, February 2005.

[PMD, 2002] cité pages 5, 19
PMD, June 2002. `http://pmd.sourceforge.net/`.

[Pressman, 2001] cité pages 4, 5, 99
Roger S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill Higher Education, 5^{th} edition, November 2001. ISBN: 0-07-249668-1.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Prieto-Díaz, 1990] cité page 36
Rubén Prieto-Díaz. Domain analysis: An introduction. In Peter G. Neumann, editor, *Software Engineering Notes*, 15(2):47–54. ACM Press, April 1990.

[Reimer *et al.*, 2004] cité page 19
Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: smart analysis based error reduction. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 243–251, New York, NY, USA, 2004. ACM Press. ISBN: 1-58113-820-2.

[Riel, 1996] cité pages 4, 13, 16, 27, 36, 38
Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Rouane-Hacene, 2007] cité page 79
M. Rouane-Hacene. *Étude de l'analyse formelle dans les données relationelles — Application à la restructuration des modèles structuraux UML*. Phd thesis, DIRO, Université de Montréal, 2007.

[Sahraoui *et al.*, 1999] cité pages 22, 28, 90
Houari A. Sahraoui, Hakim Lounis, Walcélio Melo, and Hafedh Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engg.*, 6(4):387–410, Hingham, MA, USA, 1999. Kluwer Academic Publishers.

[Sahraoui *et al.*, 2000] cité pages 26, 72
Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 154, Washington, DC, USA, 2000. IEEE Computer Society. ISBN: 0-7695-0753-0.

[SemmleCode, 2007] cité page 19
SemmleCode, October 2007. `http://semmle.com/`.

[Simon *et al.*, 2001] cité pages 17, 23
Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*, page 30, Washington, DC, USA, 2001. IEEE Computer Society. ISBN: 0-7695-1028-0.

[Slinger, 2005] cité pages 17, 27
Stefan Slinger. Code smell detection in eclipse. Master's thesis, Delft University of Technology, 2005.

[Smith et Williams, 2002] cité page 12
Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, Boston, MA, USA, 2002. ISBN: 0201722291.

[Snelting et Tip, 2000] cité pages 23, 96
Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22:540–582, 2000.

[Foutse Khomh et Guéhéneuc, 2008] cité page 104
Foutse Khomh and Yann-Gaël Guéhéneuc. Do design patterns impact software quality positively? In Christos Tjortjis and Andreas Winter, editors, *Proceedings of the 12^{th} Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, April 2008. Short Paper.

[Tahvildari et Kontogiannis, 2004] cité page 26
Ladan Tahvildari and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance*, 16(4-5):331–361, New York, NY, USA, 2004. John Wiley & Sons, Inc.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Tate et Flowers, 2002] cité page 12
Bruce A. Tate and Braden R. Flowers. *Bitter Java*. Manning Publications, 2002. ISBN: 193011043X.

[The Refactory Inc., 1999] cité pages 25, 28
The Refactory Inc. Refactoring browser, October 1999. `http://st-www.cs.uiuc.edu/users/brant/Refactory/`, `http://www.refactory.com/RefactoringBrowser/`.

[Thomas, 2002] cité page ix
Dave Thomas. Reflective software engineering – From MOPS to AOSD. In Richard Wiener, editor, *Journal of Object Technology*, 1(4):17–26. ETH Zürich, September 2002.

[Tiberghien *et al.*, 2007] cité page 13
Alban Tiberghien, Naouel Moha, Tom Mens et Kim Mens. Répertoire des défauts de conception. Rapport technique Technical Report 1303, Department of Computer Science and Operations Research, University of Montréal, November 2007.

[Tichelaar *et al.*, 2000] cité page 24
Sander Tichelaar, Stephane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of ISPSE '00 (International Conference on Software Evolution)*, pages 157–167. IEEE Computer Society Press, 2000.

[Tilley *et al.*, 1994] cité page 17
Scott R. Tilley, Kenny Wong, Margaret A. Storey, and Hausi Müller. Programmable reverse engineering. *Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[Tonella et Antoniol, 1999] cité page 23
Paolo Tonella and Giulio Antoniol. Object oriented design pattern inference. In Hongji Yang and Lee White, editors, *Proceedings of the* 7^{th} *International Conference on Software Maintenance*, pages 230–240. IEEE Computer Society Press, August 1999.

[Tourwé et Mens, 2003] cité pages 19, 20, 25
Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of* 7^{th} *European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 91–100. IEEE Computer Society Press, March 2003. ISBN: 0-7695-1902-4.

[Travassos *et al.*, 1999] cité pages 4, 17
Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Proceedings of the* 14^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–56. ACM Press, 1999.

[Trifu et Dragos, 2003] cité page 22
Adrian Trifu and Iulian Dragos. Strategy-based elimination of design flaws in object-oriented systems. In Serge Demeyer, Stéphane Ducasse, and Kim Mens, editors, *Proceedings of the* 4^{th} *international Workshop on Object-Oriented Reengineering*. Universiteit Antwerpen, July 2003.

[van Emden et Moonen, 2002] cité page 17
Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society Press, October 2002.

[Wagner, 1973] cité page 75
Hans Wagner. Begriff. In H. Krungs, H.M. Baumgartner, and C. Wild, editors, *Handbuch Philosophischer Grundbegriffe*, pages 191–209. Kösel, München, 1973.

[Wake, 2003] cité pages 39, 41
William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN: 0321109295.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

[Webster, 1995] cité pages 16, 27, 36
Bruce F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, 1^{st} edition, February 1995. ISBN: 1558513973.

[Wirfs-Brock et McKean, 2002] cité pages 13, 38
Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities and Collaborations*. Addison-Wesley Professional, 2002. ISBN: 0201379430.

[Wuyts *et al.*, 1999] cité page 19
Roel Wuyts, Kim Mens, and Theo D'Hondt. Explicit support for software development styles throughout the complete life cycle. Technical report Vub-Prog-TR-99-07, Programming Technology Lab, Vrije Universiteit Brussel, April 1999.

[Wuyts, 1998] cité page 25
Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *Proceedings of the 26^{th} Conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

[Xref-Tech, 2000] cité pages 25, 28
Xref-Tech, June 2000. `http://www.xref-tech.com/xrefactory/main.html`.

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Listes

Table des figures

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

Thèse de Naouel Moha, Lille 1, 2008

Liste des tableaux

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl

© 2008 Tous droits réservés.

http://www.univ-lille1.fr/bustl