Université de Montréal

IDENTIFICATION ET LOCALISATION DES PRÉOCCUPATIONS FONCTIONNELLES DANS DU CODE LÉGATAIRE JAVA

par
Amal El Kharraz

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

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Juin, 2011

© Amal El Kharraz, 2011.

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée:

IDENTIFICATION ET LOCALISATION DES PRÉOCCUPATIONS FONCTIONNELLES DANS DU CODE LÉGATAIRE JAVA

présentée par:

Amal El Kharraz

a été évaluée par un jury composé des personnes suivantes:

Houari Sahraoui,	président-rapporteur
Petko Valtchev,	directeur de recherche
Hafedh Mili,	codirecteur
Stefan Monnier,	membre du jury
Daniel Amyot,	examinateur externe
Houari Sahraoui,	représentant du doyen de la FAS

Thèse acceptée le: 23 avril 2012

RÉSUMÉ

Traditionnellement, les applications orientées objets légataires intègrent différents aspects fonctionnels. Ces aspects peuvent être dispersés partout dans le code. Il existe différents types d'aspects :

- des aspects qui représentent des fonctionnalités métiers ;
- des aspects qui répondent à des exigences non fonctionnelles ou à d'autres considérations de conception comme la robustesse, la distribution, la sécurité, etc.

Généralement, le code qui représente ces aspects chevauche plusieurs hiérarchies de classes. Plusieurs chercheurs se sont intéressés à la problématique de la modularisation de ces aspects dans le code : programmation orientée sujets, programmation orientée aspects et programmation orientée vues. Toutes ces méthodes proposent des techniques et des outils pour concevoir des applications orientées objets sous forme de composition de fragments de code qui répondent à différents aspects. La séparation des aspects dans le code a des avantages au niveau de la réutilisation et de la maintenance. Ainsi, il est important d'identifier et de localiser ces aspects dans du code légataire orienté objets.

Nous nous intéressons particulièrement aux aspects fonctionnels. En supposant que le code qui répond à un aspect fonctionnel ou fonctionnalité exhibe une certaine cohésion fonctionnelle (dépendances entre les éléments), nous proposons d'identifier de telles fonctionnalités à partir du code. L'idée est d'identifier, en l'absence des paradigmes de la programmation par aspects, les techniques qui permettent l'implémentation des différents aspects fonctionnels dans un code objet. Notre approche consiste à :

- identifier les techniques utilisées par les développeurs pour intégrer une fonctionnalité en l'absence des techniques orientées aspects
- caractériser l'empreinte de ces techniques sur le code
- et développer des outils pour identifier ces empreintes.

Ainsi, nous présentons deux approches pour l'identification des fonctionnalités existantes dans du code orienté objets. La première identifie différents patrons de conception qui permettent l'intégration de ces fonctionnalités dans le code. La deuxième utilise l'analyse formelle de concepts pour identifier les fonctionnalités récurrentes dans le code.

Nous expérimentons nos deux approches sur des systèmes libres orientés objets pour identifier les différentes fonctionnalités dans le code. Les résultats obtenus montrent l'efficacité de nos approches pour identifier les différentes fonctionnalités dans du code légataire orienté objets et permettent de suggérer des cas de refactorisation.

Mots clés: Fonctionnalités, réingénierie, code orienté objet légataire, refactorisation, treillis de Galois, Java.

ABSTRACT

Object oriented applications integrate various functional aspects. These aspects can be scattered everywhere in the code. There are various types of aspects :

- aspects which represent business functionalities ;
- aspects related to non functional requirements or to design concerns such as robustness, distribution, and security.

The code representing such aspects can be located in different class hierarchies. Researchers have been interested in the problem of the modularisation of these aspects and many approaches were proposed : oriented programming subjects, oriented programming Aspects and oriented programming view. These approaches offer techniques and tools for designing object oriented applications based on the composition of slices of various aspects. The main benefit of the separation of aspects is supporting reuse and maintenance. Consequently, it is well worth identifying and extracting aspects of legacy object oriented applications.

Our work mainly focuses on functional aspects. Assuming that the code of a functional aspect or a feature has a functional cohesion (dependencies between elements), we suggest methods for identifying such features from the code. The idea is to identify, in the absence of any aspect oriented paradigm, the techniques used for implementing a feature in the code. Our approach consists of :

- identifying techniques used by developers to integrate a feature in the absence of aspect oriented techniques
- characterizing the patterns of these techniques
- and developing tools to identify these patterns.

We present two approaches for the identification of the existing features in the object oriented code. The first one identifies various design patterns which integrates these

features in the code. The second approach uses the formal concept analysis to identify the recurring features in the code.

We experiment our approaches to identify functional features in different open source object oriented applications. The results show the efficiency of our approaches in identifying various functional features in the legacy object oriented, and can some times suggest refactoring.

Keywords : Features, reengineerings, legacy object oriented code, refactoring, Galois lattices, Java.

TABLE DES MATIÈRES

RÉSUMÉ . . . . . **iii**

ABSTRACT . . . . . **v**

TABLE DES MATIÈRES . . . . . **vii**

LISTE DES TABLEAUX . . . . . **xi**

LISTE DES FIGURES . . . . . **xii**

DÉDICACE . . . . . **xvi**

REMERCIEMENTS . . . . . **xvii**

CHAPITRE 1 : INTRODUCTION **1**

1.1 Contexte 2

1.2 Motivations 2

1.3 Problématique et Objectifs 6

1.4 Plan de thèse 10

CHAPITRE 2 : ÉTAT DE L'ART **13**

2.1 Programmation orientée aspects 14

2.1.1 Préoccupations versus aspects 14

2.1.2 Approches statiques de la programmation par aspects 17

2.2 Identification et localisation de fonctionnalités/features 27

2.2.1 Définition de fonctionnalité (feature) 28

2.2.2 Méthodes d'identification et de localisation des fonctionnalités (features) 28

2.2.3 Comparaison 44

2.3 Conclusion 46

CHAPITRE 3 : PRINCIPES DE NOTRE APPROCHE 49
3.1 Approches pour l'identification des fonctionnalités 51
3.2 Fonctionnalités composées avec l'héritage multiple 52
3.2.1 Principe 52
3.2.2 Cas de langages ne supportant pas l'héritage multiple 54
3.3 Fonctionnalités composées par délégation à travers un agrégat 57
3.4 Fonctionnalités composées par multiplication d'états 61
3.5 Conclusion 64

CHAPITRE 4 : ALGORITHMES D'IDENTIFICATION DES FONCTIONNALITÉS LIÉS PAR HÉRITAGE MULTIPLE ET PAR DÉLÉGATION 66
4.1 Description et algorithme d'identification des instances de l'héritage multiple dans des programmes Java 66
4.1.1 Description des cas de l'héritage multiple 66
4.1.2 Algorithme d'identification des instances de l'héritage multiple dans des programmes Java 67
4.2 Description et algorithme d'identification des instances d'agrégation dans des programmes Java 70
4.3 Conclusion 72

CHAPITRE 5 : MÉTHODES MATHÉMATIQUES POUR FOUILLES DE RÉGULARITÉ 73
5.1 Les représentations graphiques et la fouille de motifs de graphe 74
5.2 Les treillis de concepts : une alternative aux graphes 76
5.3 Principes de l'analyse formelle de concepts 78
5.3.1 Notions de base 78
5.3.2 Notions fondamentales de l'analyse formelle de concepts 80
5.4 Algorithme de décomposition sous-directe de treillis 91
5.5 Conclusion 92

CHAPITRE 6 : ALGORITHMES D'IDENTIFICATION DES FONCTIONNALITÉS PAR MULTIPLICATION D'ÉTATS 94

6.1 Identification des fonctionnalités à partir de treillis facteurs 95

6.1.1 Fonctionnalités composés par multiplication d'états 95

6.1.2 Contribution et Preuves 99

6.2 L'intuition derrière notre relation binaire 104

6.3 Formalisation de la représentation pour les occurrences 111

6.4 Relation binaire et algorithme 115

6.4.1 Relation binaire 117

6.4.2 Algorithme d'identification de fonctionnalités dans un treillis de Galois 121

6.5 Application de l'algorithme 123

6.6 Conclusion 127

CHAPITRE 7 : EXPÉRIMENTATION ET VALIDATION 130

7.1 Outils 130

7.1.1 JHotDraw 131

7.1.2 JrevsersePro 131

7.1.3 JavaWebMail 132

7.1.4 FreeMind 132

7.1.5 Lucene 133

7.2 Identification des fonctionnalités issues de l'héritage multiple et de la délégation 133

7.2.1 Héritage multiple 133

7.2.2 Délégation avec agrégation 136

7.3 Identification des fonctionnalités issues de la multiplication d'états . . . 146

7.3.1 Implémentation par héritage 147

7.3.2 Implémentation par délégation 149

7.3.3 Implémentation ad-hoc 150

7.3.4 Exemple de fonctionnalités implémentées en mode structure non similaire 151
7.4 Analyse des résultats 154
7.5 Validité 155
7.6 Conclusion 156

CHAPITRE 8 : CONCLUSION 158
8.1 Résumé 158
8.2 Contributions 161
8.3 Extensions et perspectives 163

BIBLIOGRAPHIE 165

LISTE DES TABLEAUX

5.I La table du contexte $\mathscr{K}$. 84
5.II Exemple de table de contexte 89
5.III Relations flèches appliquées sur la table 5.II 89

7.I Liste des systèmes étudiés 131
7.II Les candidats de "l'héritage multiple" 134
7.III Candidats de délégation à travers un agrégat pour JHtoDraw 5.2 138
7.IV Candidats de délégation à travers un agrégat pour JavaWebMail 142
7.V Candidats de délégation à travers un agrégat pour JreversePro 142
7.VI Candidats de délégation à travers un agrégat pour FreeMind 143
7.VII Candidats de délégation à travers un agrégat pour Lucene 144
7.VIII Données expérimentales sur les différents logiciels. 147

LISTE DES FIGURES

1.1 Classes composées de différentes fonctionnalités [80] 4
1.2 Notre démarche globale. 9
1.3 Implémentation de l'héritage multiple (a) et de la délégation (b) 9
1.4 Exemple d'implémentaion ad-hoc 10

2.1 Homomorphisme entre les exigences et les programmes [103] 15
2.2 Plusieurs aspects se chevauchent dans un programme orienté objets (POO) 18
2.3 Exemples de deux sujets [109] 21
2.4 Méthodes des deux sujets de la figure2.3 23
2.5 HyperSlice et HyperModule de l'exemple de la figure2.4 24
2.6 Modèle d'un objet avec vues [102] 26
2.7 Processus de localisation en utilisant LSI [37] 32
2.8 Exemple simple d'un graphe de dépendance [32] 34
2.9 Utilisation de graphe d'appel de méthodes dans FEAT [125] 35
2.10 Exemple de localisation de concepts [119] 36
2.11 Modèle conceptuel de l'identification d'une fonctionnalité par Eisenbarth [47] 40

3.1 Exemple de multiplication d'états 50
3.2 L'implémentation de plusieurs fonctionnalités en utilisant l'héritage multiple 53
3.3 Deux fonctionnalités ayant une sous-hiérarchie commune 53
3.4 Implémentaion de l'héritage multiple 54
3.5 Exemples de l'héritage multiple dans java 55
3.6 Différents aspects fonctionnels supportés par délégation : deux patrons différents 58
3.7 Exemples supportant plusieurs fonctionnalités dans le cas de multiplication d'états 62
3.8 Exemple de patron stratégie 63

4.1 Exemple de l'héritage multiple 69
4.2 Exemple de délégation 71
4.3 Exemple d'implémentation de délégation dans java 71

5.1 Exemple d'une fonctionnalité récurrente 74
5.2 Représentation de la hiérarchie de classes de la figure 5.1 sous forme d'un graphe 76
5.3 Exemple de Diagrammes de Hasse 79
5.4 Exemple de treillis complet pour la relation "divise" 80
5.5 **Haut :**Modèle du domaine d'un compte bancaire. **Bas :** Modèle du domaine d'une carte de crédit. 83
5.6 Le treillis du contexte lié au compte bancaire. 85
5.7 Graphe dirigé correspondant aux relations flèches de la table 5.II 89
5.8 Produit direct de deux treillis 90
5.9 Décomposition sous-direct 90
5.10 Treillis correspondant à la table 5.II 92
5.11 Décomposition sous-directe de la deuxième congruence 93

6.1 La démarche de notre deuxième approche 95
6.2 Produit direct des deux hiérarchies 97
6.3 Identification des facteurs treillis qui correspondent à des facteurs de hiérarchies 97
6.4 Schéma illustrant l'identification d'un sous-treillis isomorphe au treillis initial 100
6.5 **Haut :** Exemple de produit de deux hiérarchies de classes (**Bas :**) Diagramme de classes résultant du produit 101
6.6 Treillis correspondant au diagramme de classe de figure 6.5 102
6.7 Application des relation flèches sur l'exemple de la figure 6.5 104
6.8 Cas général d'une implémentation ad-hoc 105
6.9 Exemple de patrons récurrents de classes [24] 106
6.10 Génération de treillis à partir de la relation binaire standard 107

6.11 Génération de treillis à partir de la relation binaire associée à l'héritage . 108
6.12 Génération de treillis à partir de notre nouvelle relation binaire 110
6.13 Exemple de la relation d'incidence utilisée basée sur la figure 6.8 110
6.14 Exemple de diagramme de classes 112
6.15 Représentation d'une occurrence, composée des sous-hiérarchies A et B, par deux classes racines correspondantes 113
6.16 Exemple de racines incomparables 114
6.17 Contexte simplifié basée sur la figure 6.8 115
6.18 **Gauche :** Les Classes initiales ; **Droite :** La table de contexte associée. . 116
6.19 Le treillis de concepts associé à la figure 6.18 117
6.20 Les trois occurrences de *{capabilities, schedule, assemblyLine, licenseClass}* ne sont pas indépendantes 120
6.21 Une fonctionnalité candidate peut induire plusieurs sous-fonctionnalités candidates 121
6.22 Extrait du treillis de concepts de la figure 6.21 122
6.23 **Haut :**Comment la relation binaire est générée entre classes et entre interfaces ; **Bas :**Table de contexte associée 124
6.24 **Haut :**Le treillis de concepts associé à la figure 6.23 ; **Bas :** Le treillis de concepts uniquement avec les classes et les interfaces minimales 125
6.25 **Haut :** Un exemple de diagramme de classes pour la relation d'incidence ; **Bas :** La table de contexte associée 126
6.26 Le treillis de concepts associé à la figure 6.25 avec classes minimales. . 128

7.1 Exemples de fonctionnalités supportées par héritage multiple dans JHotDraw 134
7.2 Exemples de fonctionnalités supportées par héritage multiple dans JavaWebMail 135
7.3 Division de fonctionnalité d'une classe 141
7.4 Exemple de délégation dans JHotDraw 141
7.5 Patron visiteur dans JReversePro 143

7.6 Exemple de délégation dans le cas de Lucene 145
7.7 Exemple d'héritage dans JHotDraw. 148
7.8 Autre exemple d'implémentation basée sur héritage multiple dans JHotDraw : implémentation de plusieurs interfaces. 149
7.9 Cas d'une implémentation ad-hoc de plusieurs fonctionnalités (JHotDraw) 150
7.10 Cas d'une implémentation ad-hoc dans FreeMind 151
7.11 Autre cas d'implémentation ad-hoc dans Lucene 1.4. 152
7.12 Un cas d'implémentation de fonctionnalité sous forme de multiplication d'état dans JHotDraw. 153
7.13 Un cas d'implémentation de fonctionnalité avec structure non similaire dans JHotDraw. 153
7.14 Cas de délégation dans FreeMind 154

À mes parents, surtout à mon père, sans lui je n'aurais jamais tenu le coup pour terminer cette thèse,

À mon mari,

À mes enfants pour leur donner l'exemple qu'il faut toujours persévérer,

À toute ma famille.

REMERCIEMENTS

Je tiens avant tout, à remercier mes directeurs de recherche Hafedh Mili et Petko Valtchev pour leur aide, soutien et remarques pertinentes durant toutes les années de la thèse. Sans eux, je pense que ce travail ne sera pas présenté comme tel. Leurs conseils m'ont été d'un grand recours. Toute ma gratitude pour Hafedh qui m'a soutenue tout le long de cette thèse, et m'a appris la manière comment procéder à faire la recherche, en ayant une grande patience et persévérance. Tout mon respect et ma reconnaissance à Petko pour sa rigueur et ses propositions sur comment procéder et présenter de manière formelle les notions théoriques sans oublier l'intuition derrière. J'ai eu le grand honneur d'être leurs étudiante, je pense que j'ai acquis beaucoup de connaissances dans les disciplines : génie logiciel et analyse formelle des concepts.

Je voudrais aussi remercier les membres de jury. Un très grand merci à Daniel Amyot, professeur à l'université d'Ottawa, d'avoir accepté de rapporter ce travail. Merci à Stefan Monnier, professeur à l'université de Montréal, d'avoir accepté d'examiner mon travail. Je remercie également, Houari Sahraoui, professeur à l'université de Montréal, de m'avoir accordé l'honneur d'être le président de mon jury et pour avoir accepté de représenter le doyen de la faculté des études supérieures.

Mes vifs remerciements sont adressés à mes parents qui m'ont toujours encouragé et à mon cher époux pour sa générosité inépuisable et surtout sa patience. Très reconnaissante, je tiens aussi à remercier mes enfants et à leurs dire qu'enfin je vais pouvoir leur consacrer un peu plus de temps dorénavant.

CHAPITRE 1

INTRODUCTION

Le "génie logiciel" (anglais software engineering) est une application d'une approche systématique, disciplinée et quantifiable au développement et à la maintenance du logiciel (IEEE Computer Society) [16]. Il commence par une exploration de concepts qui aboutit, après un processus de développement, à un produit exploitable [114]. À travers ce parcours, le produit traverse plusieurs phases : l'élicitation des exigences, la spécification, la conception, l'implémentation, la maintenance et finalement le retrait. Le génie logiciel s'intéresse autant aux processus et aux techniques qu'aux outils d'aide utilisés durant diverses étapes du cycle de vie du logiciel [99] [114] [129].

Dans cette thèse nous proposons une démarche pour l'identification d'aspects fonctionnels des systèmes orientés objets légataires. L'identification d'aspects fonctionnels a plusieurs avantages, elle facilite ainsi : 1) la compréhension du logiciel, 2) la maintenance et 3) si c'est possible la modernisation du code.

Nous allons définir, dans ce rapport, des approches basées sur l'analyse statique vue qu'elle a l'avantage d'être plus facile à mettre en œuvre que l'analyse dynamique au niveau de l'analyse du code et plus enrichissante vue les informations que nous pouvons extraire. Nous avons écarté l'analyse dynamique (traces d'exécution) vu que le programmeur doit avoir une connaissance des tests existants et vue la limitation du nombre de formulations de requêtes. Nous allons présenter une manière pour identifier les aspects fonctionnels qui n'exige pas une connaissance du système par l'utilisateur. Elle sera basée uniquement sur les signatures de méthodes comme source d'informations et les relations entre elles. Nous élaborons deux approches pour identifier les fonctionnalités dans le code source. Dans cette thèse, nous commençons par définir la notion de préoccupation (*concerns*) et indiquer les différents types de préoccupations existantes dans le code. Par la suite, nous analyserons plus explicitement les aspects fonctionnels, ou tout simplement fonctionnalités, ainsi que l'identification et la localisation de ces fonctionnalités. Cette localisation peut mener à détecter certains défauts de programmation ou de concep-

tion et peut mener à suggérer des possibilités éventuelles d'emballage/refactorisation de certains éléments du code légataire.

Notre travail s'intègre dans un cadre général de la maintenance de logiciel. Il vise la compréhension du code légataire complexe par l'identification des aspects fonctionnels qui le composent.

1.1 Contexte

La maintenance du logiciel (ou maintenance logicielle) correspond aux mises à jour et aux modifications apportées à un logiciel, après sa réalisation. Elle englobe l'identification des failles du logiciel et leur correction, l'amélioration de l'efficacité du logiciel ou l'ajout d'autres fonctionnalités. La maintenance comprend aussi l'adaptation du logiciel à un autre environnement [5]. On pense souvent à tort que la maintenance concerne principalement la correction des défauts (bugs). Pourtant des études indiquent que les efforts de maintenance visent principalement à améliorer les fonctionnalités des applications [94].

La maintenance pose plusieurs défis. Il faut d'abord comprendre le code au niveau de sa conception logicielle. Il faut ensuite, pouvoir identifier et localiser les parties du code qui doivent être modifiées. Dans notre cas, le but est d'identifier les parties du code qui reflètent une fonctionnalité donnée. Une question subsidiaire qui se pose est que pour une fonctionnalité identifiée, comment la modifier/corriger si elle est erronée sans affecter les autres fonctionnalités. Aussi, ayant identifié et délimité cette fonctionnalité, est-il possible de la réutiliser dans d'autres contextes ou d'autres applications.

1.2 Motivations

Depuis les débuts de la programmation, les chercheurs et praticiens ont proposé différentes manières pour modulariser les logiciels. La programmation structurée était une technique populaire dans les années 70. Afin de gérer la complexité grandissante des logiciels, la programmation structurée propose pour une organisation hiérarchique du code [7, 129]. Les programmeurs décomposent leur code en modules (appelés fonctions

et procédures). L'analyse et la conception structurées tentent de modulariser les logiciels autour des fonctions ou procédures plus ou moins indépendantes, qui s'échangent des données pour accomplir une tâche, ou un processus d'affaires donné. Dans ce cas, l'unité de composition et de réutilisation est la procédure. Cette manière de faire a permis d'avoir des parties de code plus faciles à comprendre. Elle a aussi aidé à réduire la complexité de la description du système initial en le décomposant en un ensemble de sous-systèmes ou sous-programmes moins complexes, organisés en une structure plus simple.

Cependant, la conception structurée ne facilite pas l'évolution et l'extension du système. La raison pour laquelle la programmation structurée a eu un succès limité est que les techniques structurées visent les actions ou les données mais non les deux [129]. Par exemple, *calculer la moyenne des éléments d'un tableau* est une action qui opère sur le tableau (données) et retourne la moyenne de ses éléments (données). Généralement, les techniques structurées sont orientées actions et les données sont d'une importance secondaire [129].

C'est pourquoi théoriciens et praticiens se sont intéressés au paradigme orienté objets qui promet de réaliser efficacement du logiciel à la fois extensible et fiable. Le paradigme orienté objets repose sur trois notions primordiales. La notion de l'*encapsulation* qui permet de regrouper des variables, appelées les données membres, et des méthodes au sein d'une même entité nommée classe. La notion de l'*héritage* qui permet de définir une hiérarchie de classes où chaque classe fille hérite des méthodes et des données de ses super-classes. Généralement, la classe racine est une classe générique : plus on descend dans la hiérarchie, plus on spécialise cette classe. La notion du *chargement* offre la possibilité de définir plusieurs méthodes avec le même nom mais avec des paramètres différents. La méthode désirée est sélectionnée en fonction de ses paramètres réels lors de son appel.

Le paradigme orienté objets est supposé avoir l'avantage de la réutilisabilité par rapport au paradigme procédural. Il permetterait aux utilisateurs de suivre le code d'une manière plus facile et avoir une maintenance plus rapide.

Or, ce paradigme pose de nombreux problèmes au niveau des fonctionnalités pré-

sentes dans plusieurs objets. La question qui se pose concerne l'adéquation de l'orienté objets pour capturer et modulariser certains types particuliers d'exigences ou fonctionnalités. Certaines classes sont difficilement réutilisables, puisque le code correspondant à une fonctionnalité peut être dispersé dans plusieurs classes. En effet, les fonctionnalités complexes généralement ne se limitent pas à une seule classe. Au contraire, elles impliquent plusieurs classes et méthodes. La figure 1.1 montre comment différentes fonctionnalités peuvent chevaucher différentes classes.

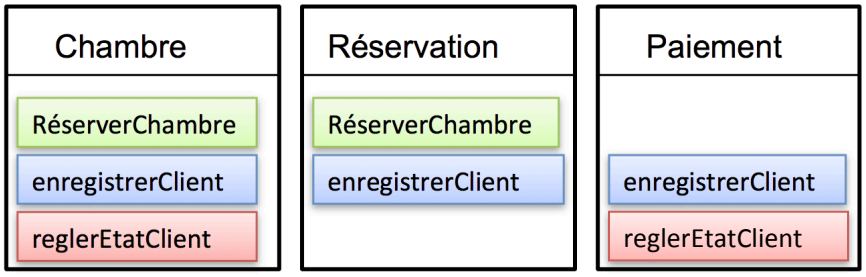


Figure 1.1 – Classes composées de différentes fonctionnalités [80]

Les techniques standards orientées objets, comme les frameworks et les patrons de conception, réduisent certains de ces problèmes, mais ne les résolvent pas tous adéquatement. De plus, la mise en œuvre de ces frameworks ou patrons de conception nécessite une phase de préplanification. Généralement, les contraintes de développement (coût, échéanciers, ressources) ne permettent pas une préplanification prévoyante de toutes les variantes possibles du logiciel.

Plusieurs nouvelles techniques de modularisation ont vu le jour. Ces techniques sont connues sous le nom de la programmation par aspects. La programmation par aspects répond à certains problèmes rencontrés dans le développement d'applications orientées objets. En particulier, elle permet de mieux structurer les applications orientées objets pour en faciliter la configuration, l'évolution, et la maintenance. Elle permet au code source à ne pas contenir des chevauchements d'aspects, ce qui le rend plus facile à comprendre et à maintenir [80]. La séparation par aspects propose un nouveau concept qui se place au-dessus de l'orienté objets pour alléger la hiérarchie des classes. Chaque aspect est implémenté séparément et peut être composé avec d'autres selon les besoins [115].

La programmation orientée aspects fournit des mécanismes pour composer les aspects à l'intérieur des opérations et les classes désirées durant la compilation ou durant l'exécution.

Les approches de programmation par aspects sont utilisées afin de rendre le code source plus lisible, facile à comprendre, plus adaptable et surtout plus facile à maintenir tout le long du cycle logiciel. La séparation par aspects permettrait aussi, d'apporter des solutions à travers l'extension du paradigme de l'objet. Elle prend en compte la notion d'aspect dans de nouvelles entités. La séparation par aspects est un élément clé dans le génie logiciel. Elle a l'habilité d'identifier, d'encapsuler et de manipuler seulement les parties du logiciel qui sont pertinentes à un concept particulier. Différentes approches ont été proposées, telles la programmation orientée sujets [74], la programmation orientée aspects [82], la programmation orientée vues [101] et les filtres de composition [18]. Il existe aussi l'approche par aspects dynamiques qui permet la modification d'un programme au moment de l'exécution (elle est utilisée pour ajouter une fonctionnalité aux composants existants dans le format binaire). AspectWerkz [4] et JBossAOP [31] sont parmi les plateformes qui supportent une telle approche pour introduire les aspects à l'exécution (run-time).

La programmation orientée sujets propose de modéliser les applications en terme de sujets ou de fonctionnalités [74] [108] [140]. Elle est basée sur la notion de sujets qui sont des *aspects fonctionnels (fonctionnalités)* relatifs (ves) aux mêmes entités. Ainsi, une application qui intègre un ensemble de fonctionnalités (sujets) sera construite comme la "*fusion*" de hiérarchies de classes, où chacune représente une fonctionnalité [74] [108] [107].

La programmation orientée aspects vise des aspects non fonctionnels. Elle permet de tisser les aspects structurels et architecturaux au code de l'application [52, 53, 82].

La programmation orientée vues offre la possibilité de connecter et de déconnecter des tranches fonctionnelles (des vues), reflétant le changement de rôles d'un objet au cours de sa durée de vie [100] [101].

Les filtres de composition sont utiles pour abstraire les communications à travers les objets. Ils manipulent les messages reçus et envoyés par les objets [18] [26].

Notre convention est qu'un aspect fonctionnel (ou fonctionnalité) peut être un sujet au sens Ossher [74]. Dans cette thèse, nous nous intéressons à identifier les fonctionnalités dans les applications légataires orientées objets. Ces applications ont été conçues et implémentées sans les nouvelles techniques de programmation par aspects et donc, plusieurs fonctionnalités y sont amalgamées. C'est pourquoi, les applications légataires s'annoncent difficiles à comprendre, et donc difficiles à maintenir [108]. Les fonctionnalités existantes sont difficiles à localiser, à tester et à modifier. Aussi, il est délicat d'intégrer de nouvelles fonctionnalités ou de pouvoir réutiliser le code correspondant aux fonctionnalités déjà existantes.

Pour ces raisons, nous croyons qu'il est important d'analyser les applications légataires qui combinent plusieurs fonctionnalités, dans le but d'isoler et d'extraire des artéfacts ou modules qui traitent les fonctionnalités individuelles.

1.3 Problématique et Objectifs

Les approches de programmations par aspects citées ci-dessus, viennent répondre à certains des problèmes rencontrés dans le développement d'applications orientées objets. Cependant, en l'absence de ces approches par aspects, un concepteur peut avoir recours à des patrons plus ou moins structurés pour implémenter et composer des fonctionnalités récurrentes dans différentes parties du logiciel.

Le but de ce travail est de développer des techniques et des outils pour l'identification des fonctionnalités dans des applications légataires. La plupart de ces applications vont intégrer plusieurs fonctionnalités, amalgamées dans les mêmes hiérarchies de classes. Nous croyons qu'au sein de ces hiérarchies de classes, l'ensemble des éléments qui constituent des fonctionnalités vont exhiber, entre elles, une plus grande "cohésion interne" qu'un ensemble d'éléments pris au hasard. De même, deux ensembles correspondants à deux fonctionnalités différentes vont exhiber un plus faible "couplage" que deux ensembles d'éléments pris au hasard. L'objectif de cette thèse est d'identifier ces fonctionnalités. L'identification des fonctionnalités permettra de comprendre le code plus facilement. L'évolution alors est plus simple et elle est à moindre coût. De même,

la réutilisation de ces fonctionnalités est plus facile.

Nous définissons une fonctionnalité par un ensemble cohésif d'exigences fonctionnelles [117] [144]. Dans un programme orienté objets, la correspondance entre les fonctionnalités et le code source n'est pas toujours explicite. Généralement, l'implémentation des fonctionnalités chevauche plusieurs classes du logiciel. Cela rend plus difficile l'identification des classes et des méthodes qui implémentent une fonctionnalité particulière. Les développeurs ont besoin d'effort additionnel pour comprendre comment les fonctionnalités sont implémentées et comment elles sont reliées entre elles. L'idée est d'identifier les techniques, en l'absence de celles des programmations par aspects, qui permettent l'implémentation des différentes fonctionnalités dans un code objet. Notre approche consiste à :

- identifier les techniques utilisées par les développeurs pour intégrer une fonctionnalité en l'absence des techniques orientées aspects
- caractériser l'empreinte de ces techniques sur le code
- et développer des outils pour identifier ces empreintes.

La hiérarchie de classes va souvent refléter une fonctionnalité (ou un ensemble de fonctionnalités) dominante(s), ce que Tarr et al. [137] appellent "décomposition dominante", alors que d'autres se trouveront éparpillées entre les classes. Dans notre travail, nous caractérisons l'"empreinte" d'implémentation d'une fonctionnalité par la manière dont cette fonctionnalité a été intégrée dans le code. Notre travail consiste principalement à identifier la manière ou la technique de réutilisation des fonctionnalités dans le code. Ces techniques dépendent de plusieurs facteurs, tels la compétence du concepteur et son niveau d'expertise ainsi que la nature du langage de programmation utilisé (par exemple le langage supporte t-il l'héritage multiple ?). Elles peuvent se manifester d'une manière assez développée et mure telle sa représentation à travers un patron de conception développé tel le modèle MVC (Modèle-Vue-Contrôleur) ou tout simplement à travers une interface au sens Java. Une autre manière moins développée et plus ad-hoc qui peut se manifester par des ensembles d'attributs/méthodes qui apparaissent

dans différents endroits d'une hiérarchie de classes, un peu à la manière du «clonage du code».

La figure 1.2 représente une vue globale de notre démarche. Notre première technique explore le code objet en identifiant la manière par laquelle les fonctionnalités ont été réutilisées. Le concepteur, en l'absence des techniques orientées aspects, a utilisé certaines techniques de l'orienté objets pour réutiliser ses fonctionnalités. Dans cette partie, l'identification de ces techniques d'implémentations des fonctionnalités existantes se base sur la nature de la relation entre les objets participants. Dans notre cas, les mécanismes qui permettent la réutilisation des fonctionnalités dans un code orienté objets légataire, sont l'héritage et la délégation (à travers un agrégat) (voir figure 1.3 (a) et (b)). Celles-ci offrent une manière de réutiliser les fonctionnalités déjà définies dans le code. La classe ***EtudiantTempsPartiel*** possède deux fonctionnalités issues de l'héritage multiple des classes ***Employe*** et ***Etudiant***. Elle peut aussi avoir une fonctionnalité issue de l'héritage et l'autre de la "délégation" à travers un agrégat de chacune des classes ***Employe*** et ***Etudiant***. Nous verrons une définition plus rigoureuse de la délégation dans le chapitre 3. Le cas de l'héritage multiple est assez simple à détecter (pourvu que le langage le supporte ; voir chapitre 3 pour le cas de Java). Pour le cas de la figure 1.3 (b), le lecteur peut constater que « ce n'est pas trivial» (voir section 3.3). Notre première technique vise à identifier ce type de relations dans un code orienté objets légataire.

La deuxième technique vise principalement le cas où le concepteur n'a pas prévu à l'avance les techniques précédentes d'implémentation des fonctionnalités déjà existantes. Il redéfinit le code de la fonctionnalité à chaque endroit où il en a besoin un peu à la manière du «clonage du code». Nous avons appelé ce type d'implémentation "ad-hoc" (voir figure 1.4). Dans cet exemple, le concepteur définit un même ensemble d'attributs dans deux endroits différents.

Pour l'identification de ce type d'implémentation, nous avons exploré l'utilisation des techniques de regroupement (clustering) qui permettent de regrouper automatiquement les fonctionnalités apparaissant dans différents endroits dans une hiérarchie de classes. En particulier, nous avons pensé à la théorie de l'analyse formelle des concepts (AFC). L'AFC fournit une description intentionnelle de chaque regroupement qui peut

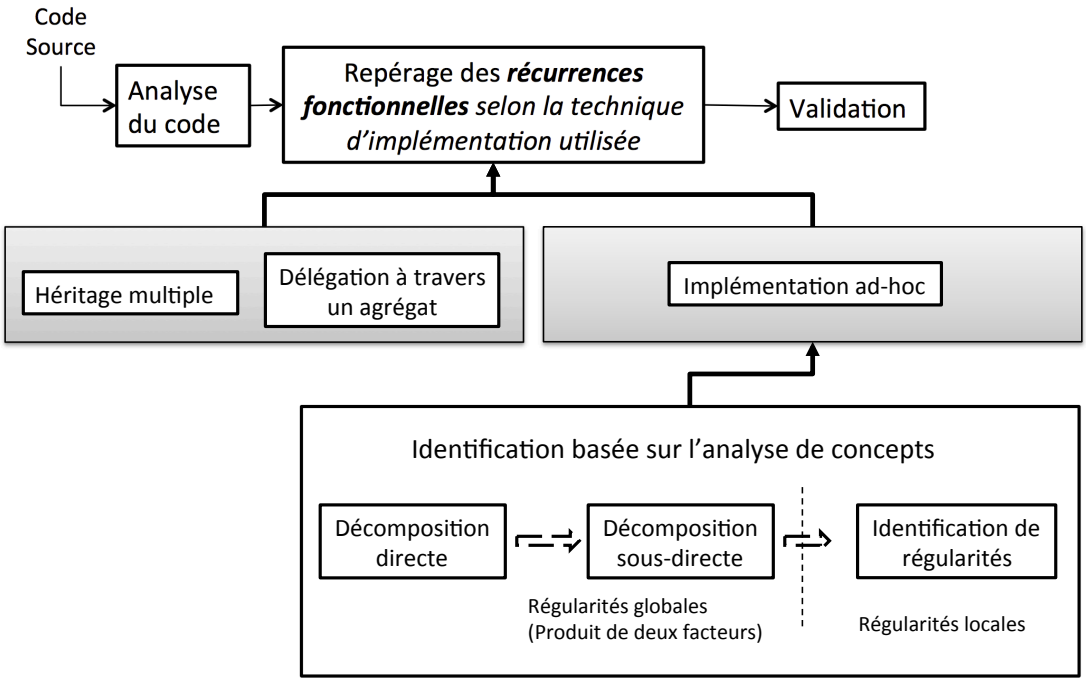


Figure 1.2 – Notre démarche globale.

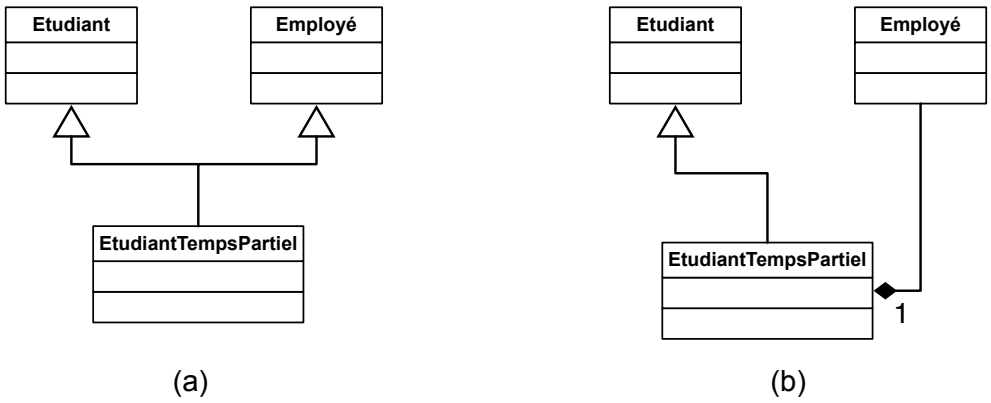


Figure 1.3 – Implémentation de l'héritage multiple (a) et de la délégation (b)

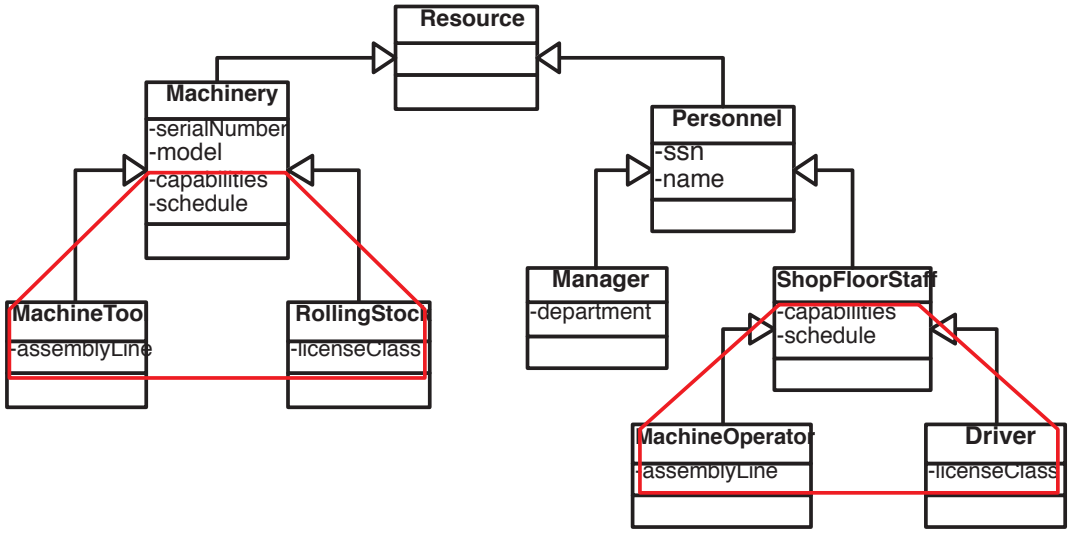


Figure 1.4 – Exemple d'implémentaion ad-hoc

rendre un groupement facilement interprétable. Les regroupements sont organisés sous forme d'un treillis, plutôt qu'une hiérarchie. L'AFC permet généralement une recherche plus riche et plus flexible. Elle est largement utilisée dans le domaine du génie logiciel pour des applications diverses [29] telles l'identification des objets dans un code légataire [14], la restructuration des hiérarchies [23, 66, 134], et d'autres [15, 132].

Ainsi, notre contribution principale est de concevoir une approche formelle qui se base sur la sémantique du code mais qui est nettement plus facile à mettre en œuvre que les approches déjà existantes basées sur l'analyse statique ou dynamique du code. En effet, on se limite aux signatures/interfaces de classes. Aussi, un même algorithme basé sur l'AFC permet de traiter tous les cas. Dans notre cas, nous avons développé une relation binaire dont le treillis résultant s'est avéré utile pour détecter autant les cas d'implantation ad-hoc, que les cas de fonctionnalités multiples implantées par délégation/agrégat.

1.4 Plan de thèse

Cette thèse est composée de trois parties majeures. La première partie englobe l'état de l'art. Elle présente une revue de la littérature sur les approches existantes pour la

localisation des fonctionnalités dans le code avec une présentation d'une synthèse comparative des approches présentées. Le fondement théorique de nos deux approches pour l'identification des fonctionnalités est illustré dans la deuxième partie. La troisième partie conclut notre thèse avec notre bilan qui résume les principales contributions ainsi que nos perspectives de recherches.

Plus explicitement :

- Dans le chapitre 2, nous présentons l'état de l'art pertinent pour notre thèse. Nous expliquons la notion de séparation des préoccupations ainsi que les différentes approches existantes basées sur ce paradigme. Nous nous intéressons particulièrement à la notion de préoccupation fonctionnelle ou fonctionnalité. Par la suite, nous détaillons les différentes techniques de localisation des fonctionnalités dans du code Java.

- Notre première approche d'identification des fonctionnalités est décrite dans le chapitre 3. Nous expliquons comment nous pouvons réutiliser les fonctionnalités à travers certains patrons. L'identification de ces patrons nous permettra par la suite de localiser les fonctionnalités réutilisées. Les algorithmes d'identification de ces patrons sont présentés dans le chapitre 4.

- Avant de pouvoir utiliser la théorie de l'analyse formelle de concepts (AFC), nous devons présenter cette théorie. Nous jugeons aussi utile de présenter et d'expliquer les différents algorithmes de décomposition de treillis pour explorer la possibilité de les appliquer. Nous suggérons qu'un treillis facteur pourrait représenter une fonctionnalité. Les détails de cette théorie sont explicités dans le chapitre 5.

- Dans le chapitre 6, nous présentons notre deuxième approche. Elle se base sur la théorie de l'AFC et utilise notre propre relation binaire entre les objets et les attributs formels. Cette relation binaire permet d'identifier différents patrons récurrents dans le code. Nous proposons que telle récurrence reflète une fonctionnalité. L'algorithme d'identification et son application sont expliqués et détaillés dans ce chapitre.

- Chapitre 7 présente la validation et l'expérimentation de nos deux approches en les appliquant sur cinq applications orientées objets en logiciel libre : JHotDraw, JavaWebMail, JreversePro, Lucene et FreeMind. Ces applications ont différents niveaux de qualité et traitent des domaines d'applications différents. Les résultats obtenus sont très prometteurs, dans la mesure où nous avons pu identifier et localiser différentes fonctionnalités issues des techniques de l'héritage multiple et de la délégation. Aussi, nous avons pu localiser les fonctionnalités issues d'une implémentation ad-hoc dans un logiciel mature tel JHotDraw.

CHAPITRE 2

ÉTAT DE L'ART

L'idée clef de la programmation par aspects est de pouvoir identifier des parties d'un programme qui reflètent un aspect donné qui joue un rôle bien spécifique. Le processus se base sur la décomposition d'un programme en aspects, où un aspect est un incrément dans un ensemble d'aspects du programme. Cette définition implique que les aspects, ne partagent aucun code avec les autres aspects. La programmation par aspects prône la séparation de l'implémentation des aspects à travers une modification des concepts existants. Cette idée a été réalisée par les différents types de programmations par aspects telles la programmation orientée aspects au sens AspectJ [81], la programmation orientée sujets au sens HyperJ [74], la programmation orientée vues [101] et bien d'autres. Ce mode d'approches a l'habilité d'identifier, d'encapsuler et de manipuler seulement les parties du logiciel qui sont pertinentes à un concept particulier. Un autre service de la programmation par aspects est l'habilité de manipuler des décompositions multiples d'un même logiciel. Ceci permet une maintenance plus facile, une meilleure réutilisation et une amélioration de la qualité du code [6].

Pour ces raisons, nous croyons qu'il est important d'analyser les applications légataires. Notre objectif dans cette thèse est de développer des approches afin de pouvoir identifier les aspects dans du code légataire orienté objets en l'absence de ces techniques de séparation. Ainsi, l'utilisateur peut utiliser le résultat obtenu afin d'améliorer la conception de son système.

Ce chapitre présente deux grands volets. La première partie définit la notion de préoccupation versus aspect, ainsi que la programmation par aspects. Nous décrirons les différentes approches de la programmation par aspects, la programmation orientée Aspects au sens AspectJ [81], la programmation orientée sujets au sens HyperJ [74] et la programmation orientée vues [101]. La deuxième partie présente le concept d'aspect fonctionnel ou fonctionnalité (*feature* en anglais). Nous décrirons ensuite les différentes techniques et approches existantes d'identification et de localisation de ces fonctionnali-

tés dans le code. Nous terminerons ce chapitre par une comparaison et une conclusion.

2.1 Programmation orientée aspects

2.1.1 Préoccupations versus aspects

Comme nous l'avons mentionné dans le chapitre de l'introduction, la «séparation de préoccupations» (*Separation of concerns, ou SOC*) est l'un des outils que les chercheurs et les développeurs du génie logiciel ont développé pour maîtriser la complexité du développement de logiciels. Cette stratégie de développement consiste à séparer les exigences de sorte à pouvoir les traiter de manière plus ou moins indépendante. Cette séparation peut se traduire par la distinction entre différentes étapes de développement, où chacune se concentre sur un ensemble d'exigences. Elle peut aussi se traduire au sein d'une même étape. Par exemple lors de la conception, on essaie de traiter ou isoler certaines exigences. Le modèle MVC permet de séparer la conception de l'aspect visualisation de la conception de la partie métier.

Une préoccupation représente une exigence alors qu'un aspect est l'implémentation de cette préoccupation/exigence quelque soit sa forme. La programmation par aspects pose le problème de développement en termes de l'existence (ou non) d'un homomorphisme entre les exigences (Requirements) et les programmes (aspects) tel qu'illustré par la figure 2.1 [103]. Les exigences sont mentionnées par $E_i(.,.)$ et les programmes par P_i. La question se pose comme suit : existe-t-il un emballage de fonctionnalité qui permette d'associer, à chaque exigence (préoccupation) un programme (aspect) qu'on pourrait composer de manière homomorphique aux exigences correspondantes ?. Il le fait en proposant trois artéfacts reliés mais distincts incarnant chacune une préoccupation distincte. Dans ce sens, chaque approche de programmation par aspect apporte sa réponse, en fonction de sa définition d'exigence (préoccupation) de son emballage d'aspects, et d'opérations de composition d'aspects.

Dans le contexte des approches orientées aspects, une préoccupation est définie comme un ensemble d'exigences reliées d'un logiciel, et un aspect est la réalisation de ces exigences. Robillard a défini une préoccupation (concern) comme suit : "*A concern is any*

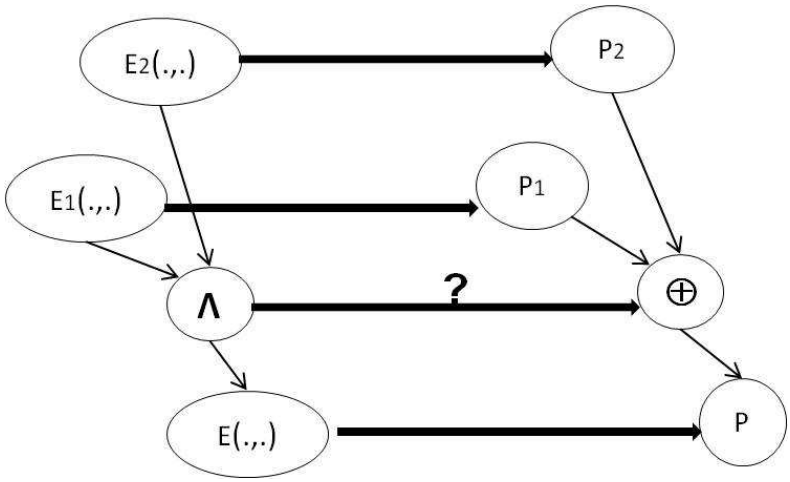


Figure 2.1 – Homomorphisme entre les exigences et les programmes [103]

consideration that can impact the implementation of a program" [122]. On distingue généralement entre deux types de préoccupations :

- Préoccupations fonctionnelles ou fonctionnalités : elles sont des préoccupations issues du domaine d'application. Elles correspondent aux exigences fonctionnelles. L'aspect correspondant à une préoccupation fonctionnelle est invoqué sur demande par le programme client. Il peut être ajouté ou utilisé selon les besoins de l'utilisateur ;

- Préoccupations architecturales : elles sont différentes des préoccupations fonctionnelles. La prise en compte de ce type de préoccupations se traduit souvent par des services d'infrastructure invoqués par défaut. Elles peuvent se relever des aspects de sécurité, de persistance, de distribution, etc. Par exemple, la sécurité d'un système peut mener à un service nécessaire d'authentification des utilisateurs à chaque utilisation d'un service.

Cependant, la mauvaise modularisation des préoccupations cause souvent des défauts de conception [44]. Une implémentation de préoccupations peut être éparpillée à travers le programme et enchevêtré dans le code source reliée à d'autres préoccupations. La préoccupation est alors dite être *crosscutting* [82]. Plusieurs études empiriques

montrent que les préoccupations éparpillées dégradent la qualité du code telle que mesurées par des métriques de qualité comme la taille d'un programme, le couplage, etc. [44]. Les préoccupations peuvent être décrites de plusieurs manières et à plusieurs niveaux d'abstraction :

- une fonctionnalité d'une liste de fonctionnalités ;
- une exigence à partir de documents d'exigences ;
- un patron de conception et éléments de conception d'un document de conception UML ;
- un style de codage tel la réutilisation de code ou l'algorithme d'implémentation utilisé, etc.

L'approche utilisée dans la conception logicielle est la séparation de ces préoccupations à travers le mécanisme de modularisation disponible dans le langage de programmation choisi [93]. Le but est d'implémenter chaque préoccupation comme un module. Hélas, les limitations des langages et des paradigmes existants combinées au manque d'expertise de conception font que l'implantation de ces préoccupations va souvent chevaucher la décomposition initiale du système [93].

Pour mieux capturer et encapsuler différentes préoccupations, tant fonctionnelles que non fonctionnelles, les chercheurs ont exploré de nouveaux moyens. Par exemple, les exigences telles le contrôle d'accès et la synchronisation entre les objets ne peuvent être exprimés dans un langage orienté objets courant sous forme de modules séparés [26]. Ces exigences peuvent être séparées grâce à la programmation par aspects. Ainsi, la séparation par préoccupations est un outil conceptuel qui permettrait mieux la modularisation.

La programmation orientée aspects (AOP) intègre les aspects dans le code d'une application (code source) de différentes manières. Une manière statique qui intègre les aspects dans le code avant le chargement de ce dernier [13]. Une manière dynamique qui vise l'intégration des aspects plus tard dans le cycle de vie, notamment, au moment du chargement ou de l'exécution. Dans ce qui suit, nous débutons par présenter

les approches statiques à la programmation par aspects. Nous détaillons les différentes possibilités du tissage des aspects. Si le tissage est exécuté avant la compilation, le tissage est statique. Nous parlerons ensuite des approches dynamiques. Elles sont basées sur l'exécution des aspects après la compilation : le tissage est alors dynamique [57].

2.1.2 Approches statiques de la programmation par aspects

Plusieurs techniques de programmation orientée aspects statiques ont été proposées dans la littérature, dont la technique qui a donné le nom à la famille, i.e. AspectJ [81], la programmation par sujets, ou implantation au sens HyperJ [3], la programmation par vues [101] et bien d'autres. Ces techniques ont des perspectives complémentaires sur ce qu'est un aspect et proposent différentes structures ou artéfacts pour les réaliser. La programmation par aspects de Kiczales et al. a connu plus de succès, et elle a bénéficié d'un ensemble d'outils matures, et d'une communauté d'usagers répandue.

Nous détaillerons, à tour de rôle, les différentes techniques de programmation citées ci-dessus en décrivant d'abord les principes puis les outils disponibles.

2.1.2.1 Programmation orientée aspects et AspectJ

2.1.2.1.1 Principe

Une préoccupation peut être réalisée de deux manières [82] :

- un composant : si elle peut être clairement encapsulée dans un objet, ou un module. Un composant est une unité fonctionnelle d'un système. Il peut être encapsulé dans une procédure généralisée (objet, méthode, procédure, API). Il est facilement localisé, accessible et facilement composable avec d'autres composants si nécessaire ;

- un aspect : si elle ne peut être clairement encapsulée dans un composant. Un aspect correspond souvent aux exigences non fonctionnelles d'un système. C'est une propriété qui affecte la performance ou la sémantique des composants du système (par exemple, l'accès à la mémoire, la synchronisation,etc).

L'implémentation des exigences non fonctionnelles pose souvent un problème avec les méthodes de programmation actuelles puisque l'implémentation se retrouve un peu partout dans les différents modules du système. Par exemple, les appels aux éléments tel que les programmes enregistreurs (loggers) sont, typiquement, dispersés un peu partout dans un programme. Ceci peut masquer la structure du programme, et en rendre la compréhension plus difficile. La mise à jour de tels aspects devient aussi difficile. C'est ce qu'on appelle des préoccupations qui se chevauchent et qui chevauchent différents modules du système (crosscuting concerns). Le but de la programmation par aspects est de permettre au programmeur de séparer clairement les aspects et les composants. La figure 2.2 présente le problème de "*cross-cutting concerns*", et montre comment la programmation par aspects permet de bien séparer les différentes préoccupations.

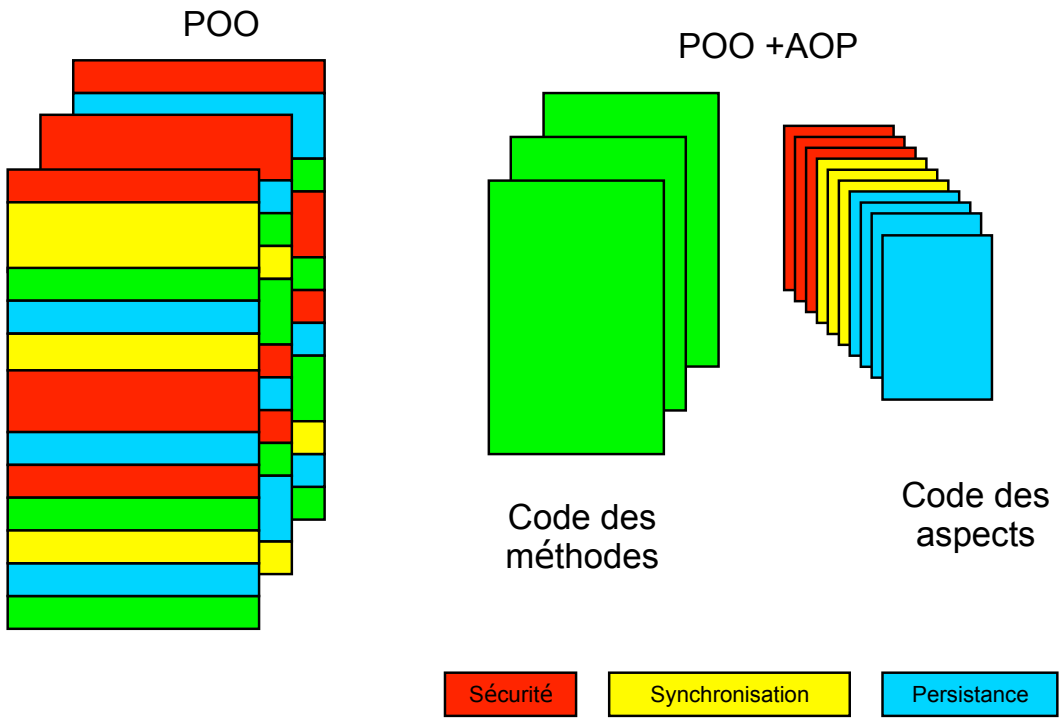


Figure 2.2 – Plusieurs aspects se chevauchent dans un programme orienté objets (POO)

La programmation par aspects [82] est une extension du paradigme objet. C'est une double structuration du code orienté objets avec les aspects. Elle compose les préoccu-

pations qui s'entrelacent avec le reste de l'application de manière modulaire. La programmation orientée aspects (AOP) apporte une solution élégante et simple à plusieurs problèmes d'intégration d'exigences non fonctionnelles. L'outil AspectJ est de qualité quasi-industrielle. Il a été utilisé dans plusieurs projets pilotes dans l'industrie [2].

AOP (Aspects Oriented Programming) consiste à décrire un logiciel comme un ensemble formé d'un composant principal et d'un ensemble d'aspects implémentant des préoccupations telles que la gestion de la mémoire, la synchronisation, les optimisations, etc. Un outil, appelé tisseur (weaver), est chargé de produire automatiquement un programme intégrant les différents aspects au composant principal. L'intérêt de cette approche est de localiser, dans les aspects, des choix de mise en œuvre qui seraient autrement dispersés dans le code source [82]. Les principes de l'AOP s'appliquent à différents langages, et différentes implémentations existantes (Java, C++, etc.). Pour la section outils, nous parlerons spécifiquement de AspectJ, l'implémentation produite par l'équipe de Xerox. PARC.

2.1.2.1.2 Outils

AspectJ est un langage pour la programmation orientée aspects [81]. Il étend le langage Java avec les aspects. Tout comme les objets, les aspects sont introduits dans la conception et l'implémentation. Durant la phase de conception, le concept des aspects nous aide à percevoir les préoccupations comme des entités à part entière que l'on peut séparer. Durant l'implémentation, le langage de programmation orientée aspects permet de programmer directement en terme d'aspects de conception, comme les langages orientés objets qui permettent de programmer directement des objets de conception [82].

Les aspects sont accompagnés de leurs règles d'intégration. Ces règles sont présentées par des points de jointures (les *joinpoint* désignent des points d'exécutions ou des évènements précis dans l'exécution d'un programme où l'aspect va être activé), des points d'actions (les *pointcuts* correspondent à la définition syntaxique d'un ensemble de points de jointure qui satisfait aux conditions d'activation de l'aspect) et des greffons (les *advices* sont similaires aux méthodes, pour spécifier un comportement à exécuter à tous les points de jointure d'un point d'action, avant, autour, après) [6].

Dans AspectJ, grâce à ses points de jointures, un aspect définit le tissage (*weaving*) nécessaire pour que la préoccupation qu'il représente soit fusionnée avec le reste du programme. AspectJ est tout simplement le langage Java avec des instructions propres à la description des aspects.

2.1.2.2 Programmation orientée sujets et HyperJ

2.1.2.2.1 Principe

La programmation orientée sujets (SOP) permet aux systèmes orientés objets d'être construits par une composition flexible de composants, appelés les sujets. Un sujet (subject) est une vue partielle du système dans le contexte du domaine d'application. Un sujet est une collection de classes qui définissent une vue particulière d'un domaine et qui fournissent un ensemble cohérent de fonctionnalités. Par exemple, un sujet peut être un ensemble de classes ou de fragments, un service, un composant ou un sous-système. Le terme fragment indique que les classes dans les sujets peuvent ne pas être complètes : elles contiennent seulement les détails dont nous avons besoin de point de vue du sujet, et non de point de vue de tout le système. La programmation par sujets distribue les objets définis par un utilisateur en une vue subjective, avec le moyen d'avoir des règles de composition entre ces objets. Ces règles de composition entre les objets correspondent à une vue d'un seul concept. Elles peuvent être décrites entre les différents éléments tels les classes, les objets, les méthodes, etc. Mais tous les éléments en relation doivent représenter une même fonctionnalité [74]. La composition de sujets intègre les classes provenant de sujets séparés, combinant plusieurs fonctionnalités. Les sujets composés forment aussi un sujet [35, 74, 108, 110].

La flexibilité de composition de sujets introduit de nouvelles opportunités pour le développement, l'adaptation et la modularisation de programmes orientés objets. La programmation orientée sujets détermine comment diviser un système en plusieurs sujets, comment intégrer les composants existants, et comment écrire les règles de composition pour pouvoir les composer correctement.

L'exemple de la figure 2.3 montre de manière concrète deux hiérarchies de classes

reflétant deux points de vue différents sur les mêmes objets. On s'imagine deux applications dans un système d'informations d'entreprise, une desservant le service de la paie, et une desservant le service du personnel [109]. La programmation par sujets permet de développer ces applications de manière séparée, et de les fusionner en une seule, au besoin. Initialement, nous avons :

- le service personnel qui gère l'information de base sur les employés ;
- le service de paye qui maintient les informations sur les salaires et les taxes des employés.

L'ensemble des classes qui définissent les fonctionnalités utilisées par le service du personnel représente un hyper slice ("hyper-tranche") encapsulant le service du personnel et le service de paye.

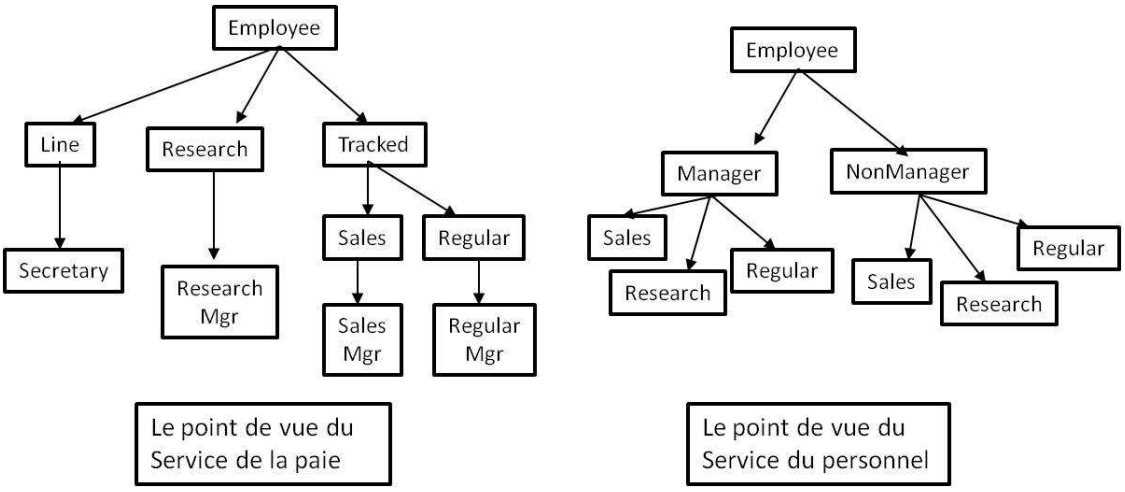


Figure 2.3 – Exemples de deux sujets [109]

2.1.2.2.2 Outils

HyperJ supporte la modularisation par préoccupations multi-dimensionnelles (*multi-dimensional separation of concerns*, ou MDSOC). Il fournit la possibilité d'identifier des préoccupations et de spécifier les modules qui traitent ces préoccupations. Il opère sur

les fichiers .class de Java sans avoir besoin des fichiers source, et produit de nouveaux fichiers .class prêts pour l'exécution. HyperJ a plusieurs avantages [3] :

- La séparation flexible et la modularisation des préoccupations : l'identification, l'encapsulation et la manipulation des préoccupations dans un programme Java standard ;
- La composition : l'extension et la configuration d'un logiciel sans en modifier le code source ;
- Le développement par équipe d'un logiciel : chaque personne peut utiliser un modèle indépendant pour sa tâche, éliminant les conflits de concurrence ;
- La traçabilité entre tous les artefacts tels les besoins, la conception et le code ;
- L'évolution : HyperJ facilite la séparation, l'utilisation et la modularisation de préoccupations, réduisant l'impact du changement dans une préoccupation sur les autres préoccupations.

Différentes préoccupations, ou opérations qui répondent à différents besoins fonctionnels, sont représentées dans la figure 2.4. Par exemple, les méthodes "position()" et "pay()", qui se trouvent dans plusieurs classes, reflètent le point de vue de la paie. Les autres méthodes reflètent le point de vue de la gestion du personnel [109].

L'outil HyperJ permet d'identifier les méthodes "position()" et "pay()" comme faisant partie d'une "tranche" (hyper-slice) que l'on peut séparer et, au besoin composer avec d'autres tranches pour donner des hyper-modules. C'est ce que les auteurs appellent la remodularisation sur demande. Un sujet doit être initialement séparé dans un module pour être composable. HyperJ nous permet d'extraire une tranche d'une application existante, et la composer avec d'autres tranches pour aboutir à des modules complets. Pour cet exemple, les méthodes "position()" et "pay()" et les classes qui les contiennent font partie de la tranche "Payroll". Alors que les autre méthodes (et les classes qui les contiennent) font partie de la tranche "Personnel". Cette association est décrite par les instructions suivantes :

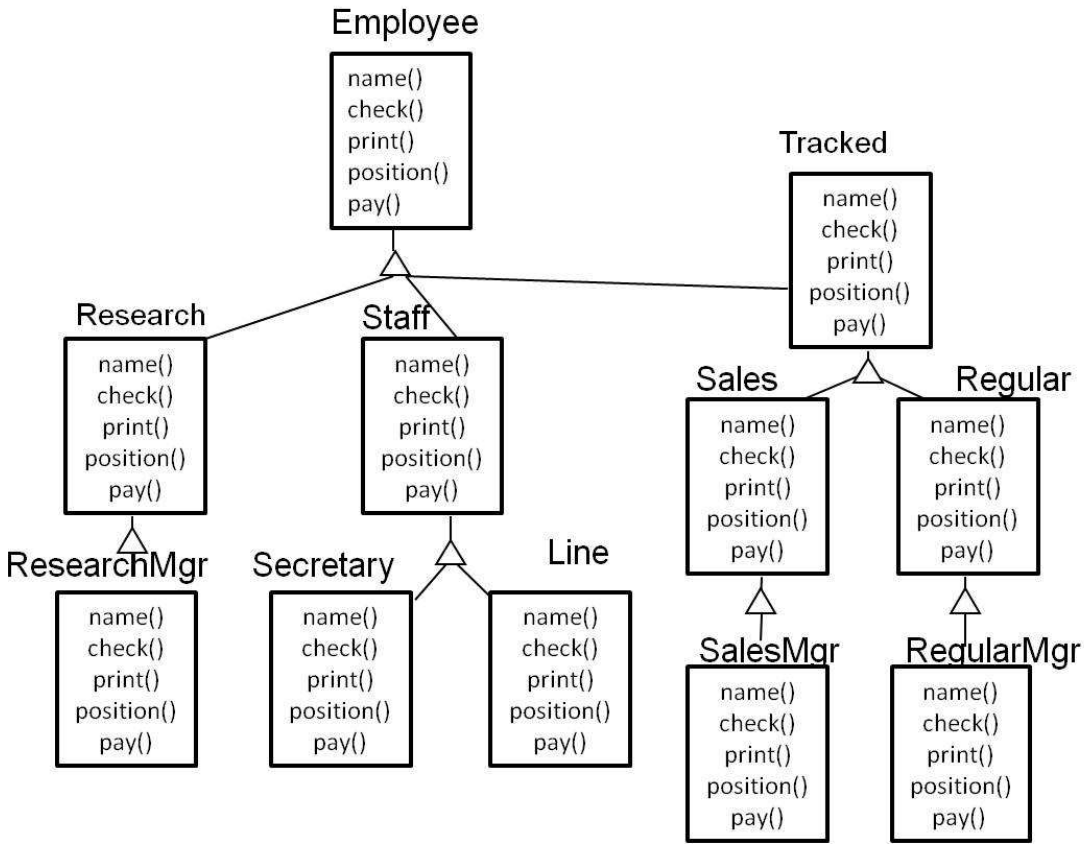


Figure 2.4 – Méthodes des deux sujets de la figure2.3

```
Package Personnel : Feature . Personnel
operation position : Feature . Payroll
operation pay : Feature . Payroll
```

La première spécifie que tout appartient, par défaut, à la tranche "Personnel". Les deux lignes suivantes viennent faire exception à cette règle, en spécifiant de manière plus spécifique les éléments qui appartiennent à la tranche "Payroll".

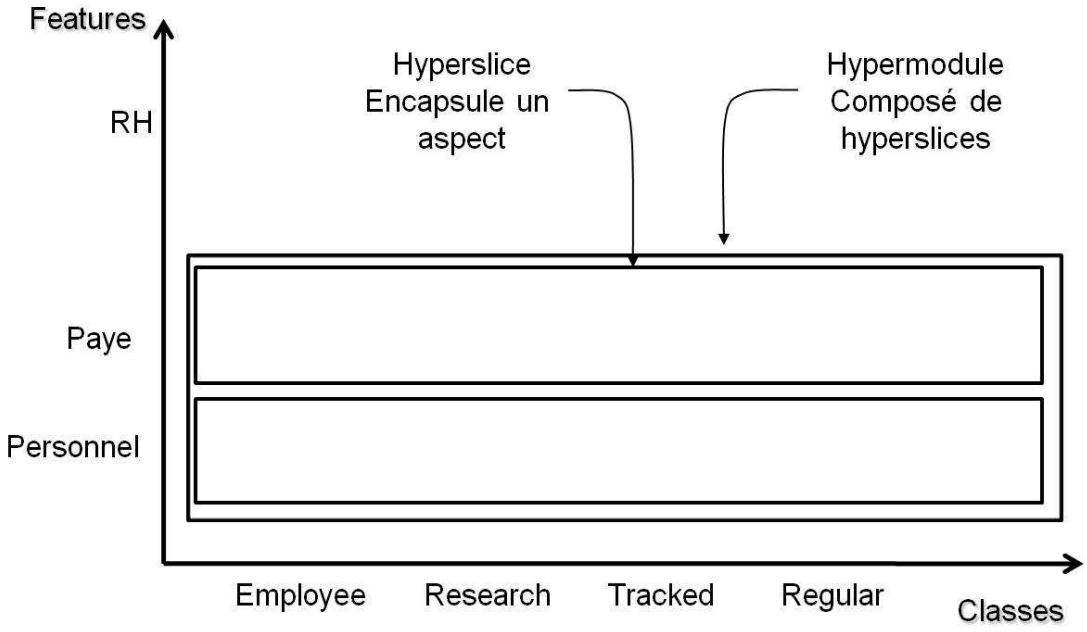


Figure 2.5 – HyperSlice et HyperModule de l'exemple de la figure2.4

La figure 2.5 illustre la distinction entre classes, hyperslices, et hypermodules. Hyper/J [3] ne permet pas d'encapsuler des protocoles de composition car les modules de composition ne possèdent ni l'héritage, ni l'agrégation. Deux tranches, comme "Payroll" et "Personnel", peuvent être composées en un hypermodule.

L'outil HyperJ permet aux développeurs de composer une collection de modules séparés appelés les hyperslices, où chacun encapsule une préoccupation en définissant et en implémentant une hiérarchie de classes appropriées pour cette préoccupation. Chaque module (qui est une hiérarchie) peut représenter une préoccupation. Il peut être aussi étendu par la composition avec autres modules.

La programmation par sujets et l'outil HyperJ ont souvent été critiqués pour plusieurs raisons. Tout d'abord, sur le plan conceptuel, la méthode ne semble pas constituer l'unité idéale de composition. Aussi, la composition des méthodes pose souvent des problèmes sémantiques épineux.

2.1.2.3 Programmation orientée vues

2.1.2.3.1 Principe

Le concept de vue dans la programmation orientée objets a été présenté la première fois par Shilling et Sweeny [79] comme un filtre de l'interface globale d'une classe. Ces vues n'étaient pas séparables ou séparément réutilisables.

La programmation par vues vise à permettre à chaque objet du domaine d'application de supporter un ensemble de fonctionnalités variables dans le temps, d'en ajouter ou d'en supprimer durant l'exécution [102]. Les interfaces peuvent correspondre à différents types d'utilisateurs ayant des intérêts fonctionnels semblables ou à différents utilisateurs ayant différents intérêts fonctionnels. Dans cette approche [101], un objet d'application se compose d'un objet noyau, et d'un ensemble variable de tranches fonctionnelles, ou des vues, reflétant le changement de rôles de l'objet au cours de sa durée de vie. L'ensemble des vues "attachées" à un objet détermine les messages auxquels l'objet peut répondre. Mili et al. [101] ont introduit la notion de point de vue (*viewpoint*) comme un "patron" ou "moule" générique permettant de décrire des vues de manière abstraite, indépendante de la nomenclature de l'objet cible. Les points de vue représentent le comportement fonctionnel d'une manière indépendante du domaine, et ils sont développés indépendamment des classes auxquelles ils s'appliquent. Ce type de programmation soutient le développement décentralisé des applications OO, et enlève plusieurs dépendances de visibilité et de propriétés que créent les techniques de développement traditionnelles [101].

La figure 2.6 montre une implémentation possible de la programmation par vues, basée sur l'agrégation. La frontière à tirets (rectangle) représente l'abstraction d'un objet d'application : elle comprend une combinaison de l'objet noyau et de ses vues. Dans cet

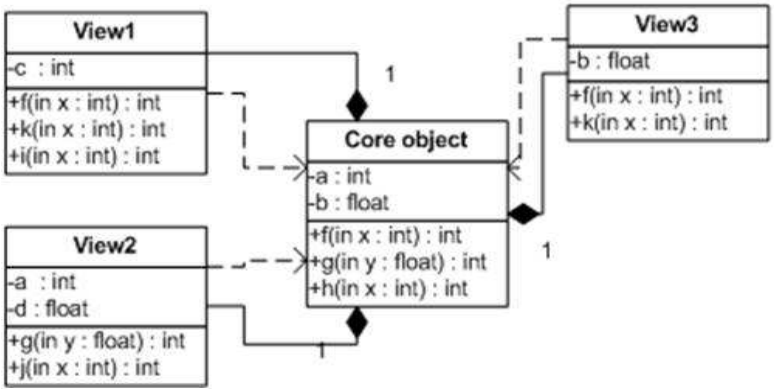


Figure 2.6 – Modèle d'un objet avec vues [102]

exemple, l'objet noyau inclut deux variables d'état ('a' et 'b'), et trois opérations (f(), g(), et h()). Les vues liées à l'objet noyau peuvent avoir des variables d'états ('c' pour vue 1 et 'd' pour vue 2), et des méthodes (i(...) pour vue 1, j(...) pour la vue 2, et le k(...) pour les vues 1 et 3). Dans ce cas-ci, en appelant l'opération f() sur la vue 1, la demande est expédiée à l'objet noyau, et la méthode f() est exécutée dans le contexte de l'objet noyau. Un fonctionnement semblable pour les références aux variables d'état partagées ('a' pour vue 2, et 'b' pour vue 3). Pratiquement, il y aura une seule copie de telles variables, stockée dans l'objet noyau, et des demandes de lecture/écriture seront expédiées à l'objet noyau. L'application est vue comme l'union des comportements d'instance de l'objet noyau et de ses vues attachées [102].

2.1.2.3.2 Outils

Les outils pour la programmation par vues ont été développés dans le cadre de travaux de doctorat d'étudiants. Ils souffrent de plusieurs problèmes :

- un manque de robustesse ;
- un manque de convivialité ;
- une couverture minimale du langage cible notamment, pour le cas de C++.

Dargham [39] a développé des outils pour supporter la programmation par vues en C++ dans le cadre de son doctorat. L'outil traite un sous-ensemble très restreint du lan-

gage C++, et s'appuie sur une transformation du code utilisateur, ce qui n'est pas toujours possible, ni très pratique.

Mcheick [98] a développé une version de la programmation par vues en Java pour profiter de la richesse et de la souplesse du langage Java. Tout en étant supérieure à la version C++, cette implantation n'est pas assez robuste ou conviviale pour être utilisée par des personnes tierces. Par contre, Mcheick a aussi développé une version de la programmation par vues pour des objets distribués, et plus précisément, des objets distribués en Corba.

2.2 Identification et localisation de fonctionnalités/features

En parallèle avec les travaux de la communauté orienté objets sur la programmation par aspects (forward engineering), les chercheurs travaillant sur la maintenance de logiciels se sont toujours intéressés à la problématique de la localisation de fonctionnalité ou «*feature*». L'identification des parties du code source correspondantes à une exigence fonctionnelle spécifique est un important processus entrepris par les ingénieurs du système durant l'évolution du projet. Les développeurs qui s'occupent de la maintenance ont besoin d'identifier les parties pertinentes du code liées à une fonctionnalité et les parties qui vont être modifiées. L'évolution du système logiciel dépend de la capacité à maintenir la localisation d'une implémentation spécifique d'une fonctionnalité dans le code source [126].

Le développeur peut ne pas avoir aucune connaissance des parties du code qui implémentent une fonctionnalité donnée ou un ensemble de fonctionnalités malgré que ce soit une information essentielle pour tout changement et mise à jour. C'est pourquoi, il est intéressant de développer des approches et des méthodes d'identification et de localisation des différentes fonctionnalités dans le code. Dans cette section, nous allons d'abord tenter de définir ce qui est une fonctionnalité ou «*feature*» (section 2.2.1). Par la suite, nous présenterons différentes approches de localisation de «*features*» (section 2.2.2). Nous concluons par une comparaison de ces approches (section 2.2.3).

2.2.1 Définition de fonctionnalité (feature)

Dans un système bien conçu, les modules doivent faire preuve d'un degré élevé de cohésion et d'un degré minimal de couplage. Chaque module s'adresse à une fonction particulière des exigences et a une interface simple [114]. La cohésion est utilisée pour mesurer la solidité fonctionnelle d'un module et pour avoir une extension naturelle du concept de dissimulation de l'information [144]. Un module cohésif doit idéalement accomplir une seule tâche. Cependant, une faible cohésion et un grand couplage induisent à un chevauchement des fonctionnalités dans le code [144]. C'est pourquoi, l'évolution et la maintenance d'un système logiciel existant exigent des programmeurs d'identifier des parties spécifiques d'un système [127] qui implémentent généralement une fonctionnalité [105] ou une activité connue [27, 28].

Une fonctionnalité est une fonction qui peut être aussi décrite dans le manuel de l'utilisateur ou une spécification d'exigence. Elle représente un ensemble d'éléments cohésifs [117] qui décrit un comportement particulier [95]. Ainsi, une fonctionnalité est une exigence fonctionnelle qui produit un comportement observable que l'utilisateur peut déclencher [48]. Le terme fonctionnalité permet aux utilisateurs de formuler les exigences [20, 47].

2.2.2 Méthodes d'identification et de localisation des fonctionnalités (features)

La localisation des fonctionnalités est une activité d'identification des unités de programme ou des éléments du code source (méthodes) qui implémentent une fonctionnalité donnée [105]. Elle est considérée comme étant une tâche parmi les plus importantes durant l'évolution et la maintenance d'un système [105]. La localisation des fonctionnalités permet d'avoir une correspondance entre les fonctionnalités et leurs implémentations. Généralement dans un programme orienté objets, la correspondance entre les fonctionnalités et le code source est non explicite. L'implémentation des fonctionnalités chevauche plusieurs modules et plusieurs couches architecturales. Cet éparpillement rend plus difficile l'identification des classes et des méthodes qui implémentent une fonctionnalité donnée. Les développeurs ont besoin d'un effort additionnel pour comprendre

comment les fonctionnalités sont implémentées et comment elles sont reliées entre elles. La localisation des fonctionnalités dans du code orienté objets existant peut être faite à travers une visualisation de la correspondance entre les fonctionnalités et le code [106].

Le but des techniques et des outils de localisation est de réduire l'espace de recherche que le développeur a besoin d'étudier et d'examiner. La plupart des approches existantes se basent sur la décomposition du code sous forme de différentes unités, autres que les fichiers, telles les classes, les fonctions, etc. Le code source est enrichi avec des informations additionnelles comme les relations entre ses éléments. La décomposition détermine l'unité de la recherche, où l'information additionnelle détermine le critère de recherche [96]. Dans des petits systèmes, les développeurs peuvent identifier les fonctionnalités manuellement. Pour les gros systèmes, ceci est plus difficile et plus compliqué. Les techniques existantes de localisation des fonctionnalités dépendent surtout de l'avis des experts du domaine pour guider le processus de localisation.

Plusieurs techniques existantes de localisation identifient effectivement le point du début de l'implémentation de la fonctionnalité, c-à-d une méthode pertinente pour une fonctionnalité [37, 91, 113]. Cependant, il est rare de trouver une seule méthode qui contribue à une fonctionnalité donnée. Pour que les approches de localisation soient efficaces, il est possible de trouver les implémentations "*quasi-complètes*" des fonctionnalités [120]. Le terme "*quasi-complète*" signifie des implémentations partielles et proches à l'ensemble des méthodes qui implémentent une fonctionnalité [120]. La connaissance de toutes les méthodes qui implémentent une fonctionnalité est plutôt subjective [121].

Il y a différentes sortes de techniques de localisation : statique, dynamique et hybride (combinaison des deux). La technique statique est basée sur toute information qui peut être extraite du programme sans l'exécuter [96]. La technique d'analyse dynamique utilise des informations recueillies lors de l'exécution [142]. Elle est basée sur la collecte et l'analyse des traces d'exécution et leurs correspondances avec le code source [20, 45, 48, 105, 133, 138, 144]. La manière hybride utilise le résultat de l'analyse statique pour progresser et diriger le processus des traces d'exécutions qui correspondent à une fonctionnalité particulière. Plusieurs outils ont été réalisés pour supporter la localisation des fonctionnalités tels ceux élaborés par Antoniol [20], Wong [144] et Wilde [105].

2.2.2.1 Analyse statique

En général, les techniques statiques de localisation des fonctionnalités sont structurelles ou textuelles. L'analyse structurelle s'intéresse à la structure sémantique du code. Par exemple, certaines approches se basent sur l'invocation de méthodes telle qu'illustrée par un graphe de dépendance statique (PDG) [75, 124]. Des approches structurelles ont été proposées par Chen [32] et Kothari [84], entre autres. L'analyse textuelle se base sur l'identification de similarité sans se soucier de la structure du programme. Les approches textuelles utilisent la technique comme le dépistage d'information [97, 113], l'analyse des composants indépendants [71] et l'utilisation d'un langage naturel [131].

Plusieurs outils utilisent à la fois l'information structurelle et textuelle pour étudier le code. Ils utilisent l'information textuelle pour élaguer les relations structurelles ou vice versa [75, 147]. Une combinaison de l'information lexicale et de l'information structurelle a été proposée par Shao [130] pour assister la localisation. Elle utilise la technique LSI (Latent Semantic Indexing) combinée au graphe d'appel avec la génération de certains cas de tests. Cette approche aide à identifier les parties du code source correspondantes à une fonctionnalité spécifique définie dans le programme. Elle peut aider aussi à localiser les problèmes de défauts de conception (bugs) dans un projet logiciel de grande taille [130].

Dans ce qui suit, nous allons présenter les différentes approches basées sur l'analyse statique telles les techniques de l'analyse textuelle suivies des techniques de l'analyse structurelle.

2.2.2.1.1 Analyse textuelle

Une des techniques les plus anciennes et les plus répandues de recherche et de localisation dans le code disponibles aux programmeurs est l'approche "pattern matching". Cette technique est supportée par une famille d'outils Unix/Linux tels que grep, egrep, fgrep, ed, sed, awk, and lex. Elle permet la recherche de motifs dans le code source d'un logiciel à travers le "pattern matching" des chaînes de caractères. Les environnements de développement modernes, tels Eclipse, rajoutent plusieurs possibilités d'un simple

pattern matching incluant les références d'une classe, noms de méthodes, etc. Ces techniques dépendent beaucoup des connaissances du développeur. Dans ce cas, une bonne compréhension du code peut mener à une bonne formulation de requêtes.

Une amélioration significative sur la correspondance d'expressions régulières est apportée par les approches basées sur la recherche d'information (IR : *Information Retrieval*) [36, 97]. La technique IR a été fournie comme une manière efficace pour la localisation. Elle établit une association entre les documents textes et le code source. Elle identifie un ensemble d'objets qui satisfont la demande et calcule le score relatif qui indique comment chaque objet correspond à la demande. Les objets sont alors triés par leur score. Une liste d'objets ayant le plus grand score est retournée aux utilisateurs. Elle permet des requêtes plus générales et liste les résultats de ces requêtes. Les approches basées sur la recherche d'informations (IR) [36, 97] localisent l'information sémantique existante dans les commentaires et les identificateurs du code source. Les systèmes IR sont utilisés pour enregistrer, gérer et rechercher les données (habituellement non structurées). Ils sont largement utilisés dans les librairies, la recherche dans le web, etc. L'inconvénient de cette approche est d'identifier la valeur choisie pour le score relatif.

Comme une partie importante du domaine de connaissances se trouve dans les commentaires et les identificateurs présents dans le code source, l'utilisation des techniques IR avancées, tel les LSI (*Latent Semantic Indexing*), permet à l'utilisateur d'établir les relations entre les termes (mots) et les documents dans des grands textes, voir la figure 2.7. LSI a été largement utilisée par plusieurs chercheurs. Deerwester et al. [40] a prouvé son efficacité en l'actionnant à partir de l'usage et non à partir du dictionnaire. Cependant, la validation a été appliquée sur deux grandes bases de données et non sur des programmes.

Marcus et al. [97] ont utilisé LSI pour la localisation des fonctionnalités pour permettre aux développeurs de formuler leurs demandes en utilisant le langage naturel. L'approche fournit une liste de résultats triés selon la pertinence avec la demande. Marcus et Maletic [10] localisent les points de départ où commence le chemin d'indexation sémantique LSI (latent semantic indexing) et utilise les techniques de dépistage d'informations IR (Information Retreival) pour identifier les correspondances entre les termes

utilisés dans une spécification d'exigences et les identificateurs dans le code. Une approche similaire à celle de Marcus a été proposée par Antonio et al. [19] pour établir les liens de traçabilité entre le code source et les documents d'exigences. Le jugement de l'utilisateur reste nécessaire pour la formulation des demandes.

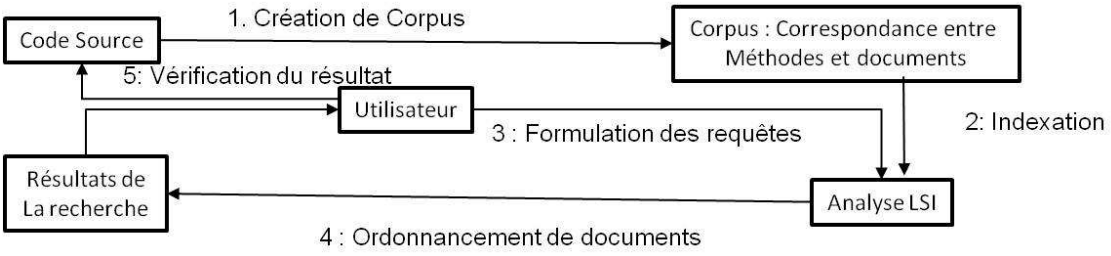


Figure 2.7 – Processus de localisation en utilisant LSI [37]

Il existe la possibilité de combiner l'approche textuelle LSI (*Latent Semantic Indexing*) avec FCA (*Formal Concept Analysis*) afin d'avoir de meilleurs résultats. Telle combinaison a été décrite par Poshyvanyk et Marcus [113]. LSI est utilisée pour identifier les fonctionnalités exprimées dans les requêtes formulées par le programmeur. Elles sont présentées sous forme d'une liste de recherche. A partir de la liste, l'approche sélectionne plusieurs attributs pertinents et organise le résultat dans un treillis de concepts généré par l'analyse formelle de concepts (FCA). L'identification des parties du code source qui correspond à une fonctionnalité spécifique est une condition préalable pour la compréhension du code. La technique la plus commune utilisée pour la localisation est la recherche dans le texte, où les développeurs écrivent des requêtes et un moteur de recherche retourne une liste d'éléments pertinents du code source à ces requêtes. Afin de réduire le coût de temps d'étudier et d'examiner la liste des résultats, les développeurs reformulent leurs requêtes pour réduire la taille de cette liste. LSI permet à l'utilisateur d'explorer le code source ainsi que la documentation textuelle reliée en formulant des requêtes en langage naturel et de récupérer une liste d'éléments pertinents (par exemple : classes, méthodes, fichiers) du code source en se basant sur leurs similarités à la requête. FCA permet une organisation automatique des résultats. Basé sur les résultats de la recherche, un treillis de concepts étiqueté (*labeled*) est généré automatiquement. Les développeurs peuvent déterminer un nœud pertinent ou non à partir du treillis, en

examinant simplement son étiquette. Les fonctionnalités sont des concepts spéciaux qui sont associés aux fonctionnalités visibles pour l'utilisateur du système. Le but partagé de ces techniques est d'identifier les unités (par exemple classes, méthodes, etc.) qui implémentent spécifiquement une fonctionnalité. Cependant, cette approche ne propose pas un processus automatisé. Comme celles qui précédent, elle se base beaucoup sur les connaissances des développeurs pour formuler et reformuler les requêtes afin de réduire les listes de recherche.

En génie logiciel, LSI a été utilisée pour une grande variétés de tâches, liées à la localisation des fonctionnalités, telle la réutilisation [145, 146], l'identification des types de données abstraits [77], la détection des concepts clonés [9], le lien de traçabilité entre les artefacts logiciels [10, 17, 19], le traçage des exigences (requirements) [41], l'identification des sujets dans le code source [87], et bien d'autres. Marcus a présenté une bonne étude sur les techniques statiques existantes dans [96].

Malheureusement, les approches basées sur les techniques IR souffrent du même problème que la technique basée sur *grep* dans la mesure de formuler une bonne requête, malgré la flexibilité utilisée dans la formulation des requêtes. La seule manière d'évaluer la qualité de la requête de parcourir la liste de résultats. L'avantage d'avoir cette liste est que l'utilisateur a un certain potentiel d'apprendre plus rapidement les informations pertinentes qu'avec grep. L'inconvénient de l'approche est qu'elle se charge uniquement de l'information lexicale dans le code source et elle ne considère pas l'information structurelle.

D'autres techniques de localisation ont été basées sur la création d'une base de connaissances ou d'un modèle du domaine. Une fois le modèle créé, il est utilisé pour localiser les concepts dans les modules du logiciel. Biggerstaff et al. [27] ont aussi considéré la notion de concept. Un concept est une partie du programme qui reflète un rôle donné. Ils ont implémenté un outil qui extrait les identificateurs à partir du code source et les regroupe pour supporter l'identification des différents concepts. Les systèmes IRENE [139], DM-TAO [27, 28] et HB-CA [69] utilisent une approche de base de connaissances. Liu et Lethbridge [92] ont réalisé une autre technique pour extraire l'information recherchée. L'idée est de construire une base de données pour enregis-

trer tous les identificateurs afin de faciliter les requêtes du programmeur. Michail [11] considère que la base pour la recherche et l'exploration est l'information enregistrée à partir de l'interface graphique GUI d'une application. Cependant, toutes ces techniques exigent une implication considérable de l'utilisateur durant la création et la maintenance des informations contenues dans la base de connaissances.

2.2.2.1.2 Analyse structurelle

L'analyse structurelle est basée sur les graphes de dépendance. Elle permet le retour de l'information structurelle telle le contrôle de flux, le flux de données, l'héritage, etc. Elle est très importante durant la recherche et l'analyse des résultats pertinents. Parmi les techniques de localisation, on retrouve celle proposée par Chen et al. [32] basée sur l'exploration de graphe de dépendance de système abstrait [32] (voir figure. 2.8). Cette technique est utile si l'analyste est familier avec le système.

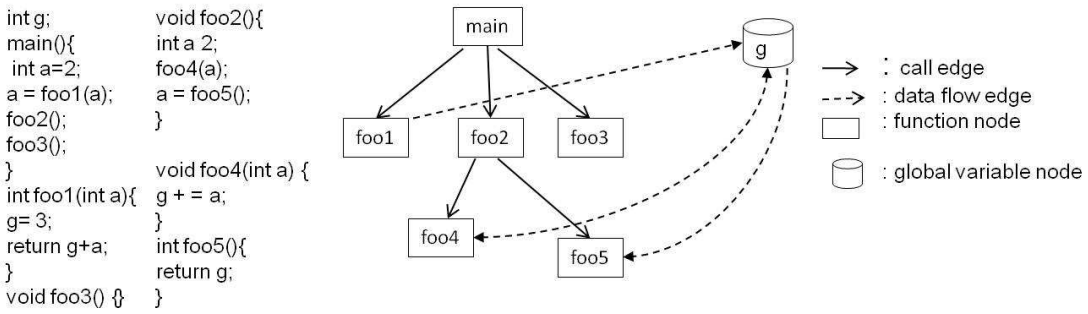


Figure 2.8 – Exemple simple d'un graphe de dépendance [32]

Une autre approche similaire a été décrite par les outils Rigi [12] et Scan (Source Code ANalyser) [112] appliqués aux programmes C. Ces outils sont basés sur le graphe d'appel afin d'extraire les artéfacts logiciels tels les types de données, les fonctions et leurs dépendances. Par suite, l'approche de recherche (ou *browsing*) représente le code source comme un graphe où les nœuds représentent des entités du système et les arcs représentent la relation entre ces entités. L'environnement est basé sur le modèle architectural hiérarchique afin de donner différentes vues architecturales du système ou des parties du système [56]. FEAT [125] et Ripples [33] sont des outils basés sur cette ap-

proche. Ripples [118] utilise le graphe de dépendance afin d'aider l'utilisateur durant le processus de changement incrémental alors que FEAT facilite la séparation en utilisant le graphe d'appel entre méthodes (voir 2.9). Celui-ci a été nommé par graphe de préoccupations (*concerns*).

Figure 2.9 – Utilisation de graphe d'appel de méthodes dans FEAT [125]

Rajlich [119] utilise aussi le graphe d'appel, mais entre les classes, afin d'identifier toutes les classes participantes à une fonctionnalité donnée (voir figure. 2.10). Il utilise la localisation de concepts où un concept est une partie du programme qui reflète un rôle spécifique.

En résumant ce qui précède, la technique de recherche basée sur la dépendance structurelle peut être utilisée sans aucun support d'outil spécifique. Il est plus facile pour l'utilisateur de suivre les dépendances manuellement à travers le code, mais uniquement pour les petits systèmes. Pour les gros systèmes, le graphe est difficilement lisible. Aussi, la technique dépend sur la capacité du programmeur à comprendre correctement les fonctionnalités des classes.

Zhao et al. [147] ont proposé une combinaison qui utilise cette fois-ci le graphe d'appel (*branch-reserving call-graph information*) [32] avec les techniques IR pour guider l'information et affecter automatiquement les fonctionnalités avec ses éléments respectifs dans le code source [147]. Cette approche a été récemment étendue par Ro-

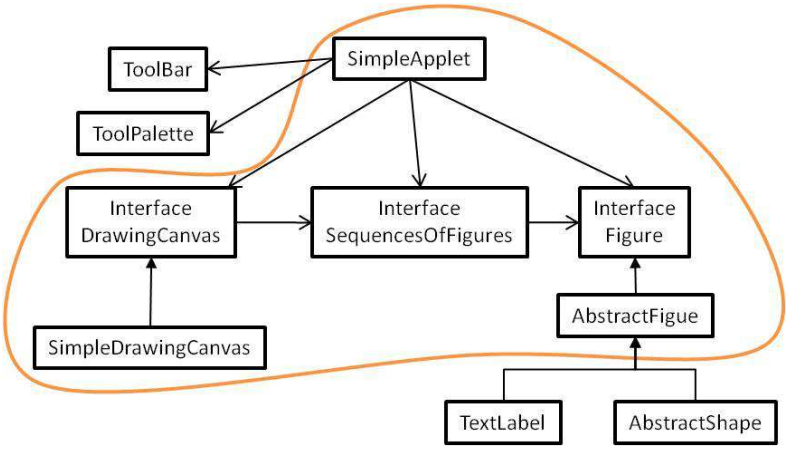


Figure 2.10 – Exemple de localisation de concepts [119]

billard [123] via l'analyse des topologies de dépendance pour ranger les éléments pertinents dans le code source. Ce dernier ne s'est pas contenté de deux combinaisons, mais a exploré différentes combinaisons variées d'analyses statique, dynamique et textuelle [121]. Les approches sont évaluées en terme de découvrir les méthodes pertinentes. Une nouvelle manière d'appliquer l'analyse textuelle est introduite par les requêtes qui sont automatiquement composées des identificateurs d'une méthode pertinente pour une fonctionnalité. Une correspondance a été établie entre un nombre d'éléments comme les attributs et les méthodes et les préoccupations existantes dans du code source. L'approche se base principalement sur les identificateurs des méthodes. Une autre approche différente des précédentes mais ayant le même but, qui est l'identification des méthodes pertinentes à une fonctionnalité, est présentée par Revelle et Poshyvanyk [120]. Elle présente aux programmeurs une liste de méthodes et leur demande de déterminer la pertinence de chaque méthode pour une fonctionnalité donnée. Pour déterminer si une méthode est pertinente pour une fonctionnalité, il suffit de vérifier [120] la similarité entre le nom de la méthode avec les mots décrivant la fonctionnalité, étudier le code de la méthode s'il est pertinent à la fonctionnalité et/ou explorer les dépendances structurelles de la méthode. Comme celle de Robillard, le point faible de ces approches est la difficulté de trouver le seuil à travers duquel les programmeurs peuvent juger la pertinence d'une méthode pour une fonctionnalité.

En résumé, nous distinguons deux types d'analyse statique : textuelle et structurelle. Il existe des approches basées uniquement sur l'une ou sur l'autre, alors que certains chercheurs, pour améliorer l'efficacité des résultats, ont procédé à la combinaison des deux. Comme cité précédemment, les inconvénients majeurs de ces analyses sont que l'analyse textuelle se base sur les informations existantes dans les commentaires et les identificateurs alors que l'analyse structurelle, se base sur les connaissances du programmeur pour la formulation des requêtes afin de limiter les listes de recherches. Une catégorie d'analyses basées sur les traces d'exécutions est présentée par ce qui suit.

2.2.2.2 Analyse dynamique

L'analyse dynamique est une approche qui se base surtout sur l'extraction de l'information dynamique et elle est basée sur les traces d'exécutions. Wilde et al. [62, 105] sont les premiers qui ont traité le problème de localisation de fonctionnalités avec la méthode de reconnaissance du logiciel en utilisant l'information dynamique.

Ils ont catégorisé les unités exécutées pour une fonctionnalité f comme suit [105] :

- code exécuté dans tous les cas de tests sans se soucier de f ;
- code exécuté dans au moins un cas de test qui invoque f ;
- code exécuté dans tous les tests qui invoquent f ;
- code exécuté exactement dans les cas où f est invoquée.

Cette approche a été étendue et améliorée par plusieurs chercheurs en utilisant de nouvelles méthodes pour analyser les traces d'exécutions [20] et comment sélectionner les scénarios d'exécutions [48]. Wilde et Scully [105] ont introduit la reconnaissance basée sur l'analyse du chevauchement des traces d'exécution des cas de tests. Cette dernière a été formalisée par Deprez et Lakhotia [42]. Pour identifier les fonctionnalités, le développeur a besoin de formuler au moins deux cas de tests. Le premier test filtre les fonctionnalités désirées alors que le deuxième filtre celles non désirées. La technique a été développée par Simmons [133] et a été adaptée pour être appliquée sur les systèmes distribués par Edwards [45]. Une autre approche similaire, basée sur les cas de

tests, a été utilisée par Wong [144]. Elle analyse l'exécution de parties du code (slices) en implémentant des cas de tests mais seulement pour cibler une préoccupation particulière. L'idée est de différencier entre l'ensemble des unités exécutées. La connaissance du système par le programmeur est nécessaire. Une comparaison des techniques complémentaires sur une étude de cas a été faite par Rajlich et Wilde [141].

Eisenberg [48] a introduit l'analyse des traces dynamiques (*Dynamic Feature Traces*). Le but principal est d'utiliser ces traces dynamiques et d'assurer qu'elles peuvent facilement être réalisées par des développeurs non familiers avec le système. Une partie du processus utilise des heuristiques afin d'énumérer les méthodes en relation avec la méthode pertinente pour une fonctionnalité. Les cas de tests sont manuellement partitionnés dans un sous-ensemble de fonctionnalités spécifiques qui sont par la suite utilisées pour obtenir les traces d'exécutions. En se basant sur les traces collectées pour ces fonctionnalités, les méthodes sont classées en utilisant des heuristiques basées sur certains critères tels l'appel entre méthodes. La différence entre cette technique et les autres techniques qui utilisent l'analyse des traces d'exécutions est qu'elle emploie graduellement des heuristiques de classement pour déterminer les éléments pertinents du code associé à une fonctionnalité à l'opposé des jugements binaires qui sont soit pertinent ou non. Les heuristiques utilisées ont un score entre 0 et 1 [48]. Un exemple d'heuristique, génère l'appel entre méthodes. Pour un ensemble de tests, elle regroupe à travers les trace d'exécutions l'ensemble des méthodes appellantes et les méthodes appelées. Un autre exemple d'heuristique, calcule la multiplicité des méthodes en identifiant les méthodes les moins invoquées pour un ensemble de tests. L'inconvénient des heuristiques est qu'elles se basent principalement sur le jugement de l'utilisateur pour identifier les parties pertinentes ou non.

En se basant toujours sur les traces d'exécutions, Bohnet et Dollner ont présenté un outil pour la localisation et la compréhension des fonctionnalités, cette fois-ci dans du code source C/C++ non familier [78]. Les concepts clefs de l'outil sont : (1) la combinaison du graphe d'appel dynamique avec le contenu des fonctions à l'intérieur des modules structurés hiérarchiquement et (2) le moyen de regrouper l'information de l'état du programme en affectant les points de contrôle (*breakpoints*) sur les localisations définis par

l'utilisateur à l'intérieur du graphe d'appel. L'outil étudie comment les artefacts interagissent pour produire une fonctionnalité. En suivant le même principe, Olszak [106] a utilisé les traces d'exécutions dans des packages Java pour localiser l'implémentation des fonctionnalités pour établir les modules explicites concernés. L'idée est de regrouper les classes qui implémentent les différentes fonctionnalités du programme en identifiant les méthodes des classes qui participent à l'implémentation de ces fonctionnalités. La représentation est établie en associant les classes aux modules des fonctionnalités. La localisation utilise les points d'entrée pour organiser l'exécution des fonctionnalités lors de l'exécution d'un programme. Un point d'entrée est la première méthode à travers laquelle un fil d'exécution (thread) est lancé quand un utilisateur interagit avec une fonctionnalité du programme. Une méthode est placée dans une trace de fonctionnalité seulement si elle est exécutée au moins une fois durant l'exécution du programme. Les points faibles de ces approches est qu'il faut identifier les bons scénarios pour identifier les fonctionnalités du système que le développeur a besoin de modifier et la nécessité d'avoir une technique d'analyse comme le débogage pour reconnaitre les informations sur l'état du programme.

Dans l'analyse dynamique, les traces d'exécutions sont souvent nombreuses et contiennent beaucoup de bruits comme observé par Antoniol [20] : *" we cannot distinguish feature-relevant and feature-irrelevant events with one unique trace alone. We need multiple traces from different scenarios and exercising different features to identify feature-relevant events"*. La sélection des cas de tests ou de scénarios lors de l'exécution est un autre problème de ces techniques. Plusieurs techniques dynamiques utilisent au moins deux traces d'exécutions, où le rôle de l'une est de pouvoir filtrer l'autre. Une autre limitation de cette approche est qu'elle génère des traces d'exécutions pour traiter plusieurs fonctionnalités du système, alors qu'une demande de maintenance peut typiquement viser une et une seule fonctionnalité particulière.

2.2.2.3 Analyse hybride

L'analyse hybride est la combinaison de l'analyse statique et dynamique. Le but est d'améliorer le processus de localisation en ayant les avantages des deux analyses [83].

Eisenbarth et al. [47] ont combiné l'information statique sous forme de graphe de dépendances et l'information dynamique sous forme de traces d'exécutions pour identifier les fonctionnalités dans les programmes et pour établir un rapport entre l'ensemble de ces fonctionnalités. Ils ont identifié les unités computationnelles qui implémentent spécifiquement une fonctionnalité (voir figure. 2.11). Ils décrivent comment classifier les routines sous forme de spécifique, pertinente, conditionnellement spécifique, partagée et non pertinente [47]. L'information dynamique est générée à partir des traces d'exécutions basées sur un ensemble de scénarios. Ils ont appliqué ensuite l'analyse formelle de concepts (AFC) pour regrouper les traces d'exécutions afin de déterminer les relations entre les fonctionnalités. L'analyse statique est utilisée pour identifier les composants pertinents à une fonctionnalité. Cette approche exige une connaissance considérable du système avec la génération de plusieurs cas de tests pour identifier une seule fonctionnalité. Il existe une correspondance entre le treillis original avec le contexte formel (fonctionnalités, scénarios) et le nouveau treillis avec le contexte formel (scénarios, unités computationnelles). Cependant, cette technique [47] a le problème de la non correspondance directe entre les fonctionnalités et les cas de tests.

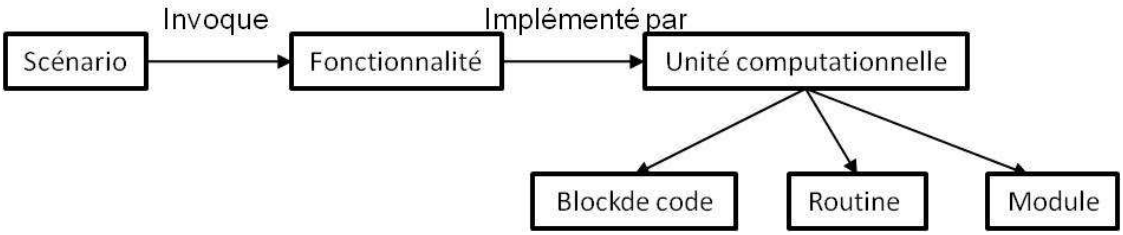


Figure 2.11 – Modèle conceptuel de l'identification d'une fonctionnalité par Eisenbarth [47]

Koschke a utilisé aussi le principe de la localisation dynamique des fonctionnalités avec l'AFC [83]. L'approche permet de localiser les unités exécutées pour un ensemble de fonctionnalités. La combinaison de fonctionnalités dans les cas de tests augmente le nombre des unités exécutées, ce qui provoque l'explosion combinatoire dans l'AFC et ne permet pas de couvrir toutes les fonctionnalités. Par conséquent, le nombre de fonctionnalités et les cas de tests manipulés restent limités. C'est la limitation de l'analyse

dynamique. En général, les approches basées sur plusieurs cas de tests rendent la tâche difficile pour la localisation dynamique de fonctionnalités, pas seulement celles qui utilisent l'analyse de concepts. Par contre, l'avantage de l'utilisation de cette dernière est de faire ressortir les fonctionnalités atomiques et les fonctionnalités composées dans une même structure.

Poshyvanyk et al. [37] ont introduit une autre approche hybride qui a l'avantage d'exiger une seule trace pour la localisation des fonctionnalités. Une trace est un ensemble d'événements regroupés dynamiquement en exécutant un scénario donné. Un événement correspond à une méthode et ses appels de fonctions. Elle est basée sur IR (information retreival) pour décrire les composants invoqués dans une trace. L'approche est nommée PROMESIR [37] et elle est l'une des plus récentes approches hybrides. Elle combine les deux techniques existantes pour l'identification des fonctionnalités : SPR (*Scenario-based Probalistic Ranking events*) [20, 60] et LSI [97]. Avec les traces d'exécutions résultantes, SPR produit un ensemble pertinent de méthodes pour une fonctionnalité. Aussi, le développeur formule une demande décrite dans un langage naturel pour localiser la fonctionnalité pertinente. En utilisant LSI, toutes les méthodes dans le système sont classées en respectant cette demande. Les auteurs ont évalués cette approche sur plusieurs cas d'études tel Mozilla et les résultats montrent que l'approche proposée améliore efficacement la localisation des fonctionnalités comparées à celles utilisant SPR ou LSI seule. Le désavantage est qu'elle est reliée à la connaissance informelle, tels les commentaires du code source et les identificateurs. Aussi, la difficulté réside dans le choix du bon scénario pour une trace donnée.

Liu [91] a présenté une technique basée sur la combinaison d'informations de deux sources différentes, traces d'exécutions combinées avec les commentaires et les identificateurs à partir du code source. Son approche est aussi basée sur l'idée qu'une seule trace d'exécution d'un scénario, pour une fonctionnalité spécifique, contient toutes les informations nécessaires pour trouver les parties importantes du code source qui implémentent cette fonctionnalité [48, 72, 85, 128]. En utilisant l'analyse textuelle, Liu filtre la donnée suffisante pour extraire les pièces pertinentes du code source. Les développeurs formulent des traces marqueurs pour réduire la taille des traces [128]. Ils

utilisent l'indexation sémantique (LSI) [40] et la méthode IR (Information Retreival). LSI est utilisée pour indexer l'information textuelle à partir du code source (commentaires et identificateurs). Elle permet à l'utilisateur d'exécuter les requêtes décrivant une fonctionnalité dans un langage naturel et obtenir les résultats sous forme d'une liste d'éléments du code source (classes, méthodes, fonctions). La méthode est nommée SITIR (*SIngle Trace and Information Retrieval*) [91]. C'est une technique qui génère un ensemble de méthodes pertinentes à partir des traces marqueurs pour une fonctionnalité désignée. La difficulté majeure est d'arriver à identifier une seule trace d'exécution d'un scénario. Aussi, comme les autres approches basées sur LSI, le point faible réside dans la bonne formulation des requêtes.

Avec la même idée d'avoir une seule trace, Rohatgi [126] a présenté cette fois-ci l'analyse dynamique illustrée par la génération d'une trace d'exécution avec le graphe de dépendance de composants (analyse statique) utilisé pour trier les composants invoqués lors de la trace suivant leur pertinence par rapport à la fonctionnalité. La technique est basée sur l'impact de modification d'un composant sur le reste du système. Comme Poshyvanyk et al. [37], l'idée est la génération d'une seule trace correspondante à une fonctionnalité [126]. À partir de la trace d'exécution, un profile d'exécution qui correspond aux différentes classes invoquées dans la trace est identifié. L'analyse de l'impact pour identifier les classes spécifiques à une fonctionnalité est mesurée par l'impact potentiel de modifications de chaque classe distincte dans le profil d'exécution sur les parties du système [126]. Plusieurs types de relations peuvent exister entre deux classes tel l'appel de méthodes, l'héritage, etc. Par contre, la difficulté réside dans la génération de la bonne trace.

Comme Liu [91], Antonio et Guéhéneuc [20] ont développé une approche basée sur les traces marqueurs (*marked traces*), mais avec l'application des tests statistiques des événements qui se présentent dans ces traces marqueurs. Elle filtre une base de connaissances et supporte les applications multithreads en utilisant des techniques pour la collection de traces dans C/C++ et Java. Un scénario fournit les conditions sous lesquelles une fonctionnalité est invoquée. Les données dynamiques sont regroupées de différents scénarios comme traces, liste d'événements composée d'instanciation d'objets, accès

aux variables, fonctions et appel de méthodes. Ils utilisent ensuite une métaphore pour classifier ces événements comme pertinent ou non pour la fonctionnalité considérée. Ils proposent des analyses statistiques des données statiques et dynamiques pour identifier les fonctionnalités dans les programmes orientés objets "multithreads". L'inconvénient des analyses statistiques est qu'elles sont approximatives et il faut savoir analyser et interpréter les résultats.

Généralement, la combinaison des différents types d'analyses a été utilisée pour améliorer le résultat obtenu. L'approche nommée Cerberus [43] utilise trois types d'analyses : l'analyse IR, l'analyse dynamique et l'analyse structurelle. Elle applique IR afin d'avoir l'ordonnancement de l'information extraite des commentaires et des identificateurs et utilise les traces d'exécutions pour avoir un ordonnancement des méthodes qui invoquent une fonctionnalité. Ensuite, elle utilise une formule appliquée aux deux listes, celle de l'information extraite et celle des méthodes. Le graphe de dépendance est utilisé ensuite pour identifier les éléments pertinents liés à une fonctionnalité.

Pour identifier les fonctionnalités, certains travaux fournissent une évidence que les approches hybrides (statique et dynamique) [37, 47] basées sur la combinaison de l'information obtenue à partir des traces d'exécutions et de l'information textuelle obtenue à partir du code source sont des techniques d'identification très efficaces [20]. Par contre, elles ont leurs propres limitations. Pour filtrer les traces d'exécutions, elles utilisent l'information statique obtenue à partir de la formulation des requêtes par les programmeurs. Aussi, le nombre des tests pour générer les traces d'exécutions reste limité.

2.2.2.4 Autres types d'approches d'identification

Il existe autres types d'approches d'identification différentes de celles présentées précédemment. Wong [54, 144] a utilisé l'approche basée sur les métriques pour évaluer la concentration d'une fonctionnalité dans un composant et l'affectation d'un composant à une fonctionnalité. Il a étendue cette approche en calculant la distance entre les fonctionnalités. Pour plus de détails, voir [54, 144]. Une autre manière d'utiliser des métriques a été présentée par Eaddy [44]. L'inconvénient des métriques est le choix du seuil pour interpréter les résultats surtout si les valeurs sont très rapprochées.

Grant [71] a isolé les sections du code source reliées à certains concepts spécifiques à partir d'une identification statistique des signaux indépendants dans le texte. L'approche est nommée analyse des composants indépendants (Independents Components Analysis). C'est une technique de séparation de signaux qui visualise les relations entre les blocs du code source. Elle sépare un ensemble de signaux d'entrées (input) qui sont mutuellement indépendants. Cette approche a les inconvénients des méthodes statistiques et la nécessité d'avoir une connaissance du système pour générer les bons signaux d'entrées.

Greevy et al [73] ont exploité les relations entre les fonctionnalités et les classes pour analyser l'évolution des fonctionnalités d'un système et de détecter les changements dans le code. Malgré l'identification des composants spécifiques à une fonctionnalité, le focus principal des auteurs de l'approche est d'étudier comment les rôles des classes changent durant l'évolution du système.

Gold et al. [70] ont proposé une approche pour les concepts reliés (bindings concepts) avec les frontières de chevauchement (*overlapping boundaries*) dans le code source. Elle est formulée comme un problème de recherche en utilisant les algorithmes génétiques et les algorithmes de recherche comme l'algorithme *hill climbing*. Une comparaison et un résumé des techniques statiques pour la localisation des fonctionnalités sont détaillés dans [96]. Plusieurs discussions concernant la localisation se trouvent dans [20].

Il existe d'autres approches qui ont ciblé l'identification de plusieurs fonctionnalités à un temps donné, comme celles qui analysent l'interaction des fonctionnalités [46, 128], l'évolution des fonctionnalités [72] et les dépendances cachées entre les fonctionnalités [55].

Par contre, toutes ces approches (celle de Greevy, Gold et les dernières approches citées) supposent que le programmeur a une connaissance de toutes les fonctionnalités du système au préalable.

2.2.3 Comparaison

En résumé, il existe plusieurs types d'analyses qui peuvent être combinées entre elles [120]. Habituellement, ces techniques exigent une connaissance du système par les

utilisateurs afin de spécifier les parties du code source qui ont besoin d'être analysées :

- Analyse statique : Celle-ci utilise des informations présentes dans le code sans recourir à son exécution [28, 123, 131, 147]. Elle peut être textuelle ou structurelle :
  - Analyse textuelle : une approche pour localiser les fonctionnalités. Elle détermine les similarités textuelles entre une demande formulée et des éléments du code source (méthodes) en utilisant des techniques de dépistage de l'information, dont le LSI [10, 17, 40, 41, 97, 113] ;
  - Analyse structurelle : elle fournit l'information sur différents types de dépendances dans le système. Elle se concentre sur l'invocation de méthodes dans un graphe de dépendance statique. (PDG) [32, 75, 118, 119, 124, 147].
- Analyse dynamique : pour collecter l'information dynamique, les utilisateurs exécutent des scénarios qui déclenchent une fonctionnalité [20, 42, 48, 105, 133]. Un scénario est une séquence d'entrées de l'utilisateur au système. Wilde [105] a été le premier qui a développé la méthode de reconnaissance, laquelle utilise les traces d'exécutions pour localiser les fonctionnalités dans des systèmes existants. Il y a deux types de traces, traces complètes et traces marqueurs. Les traces complètes (*full traces*) [105] capturent tous les événements au lancement du système jusqu'à sa fin. Les traces marqueurs (*marked traces*) [91, 128] capturent seulement les événements durant une partie de l'exécution du système. L'utilisateur peut commencer et arrêter le traçage selon sa volonté.

Les points faibles des techniques statiques [96] sont présentés comme suit :

- La technique basée sur *grep* dépend beaucoup des connaissances du développeur. Dans ce cas, une bonne compréhension du code peut mener à une bonne formulation de requêtes ;
- La technique basée sur IR souffre du même problème que la technique basée sur *grep* pour formuler une bonne requête malgré l'existence d'une bonne flexibilité. La seule manière d'évaluer la qualité de la requête est d'évaluer la liste de résultats.

L'avantage d'avoir la liste est que l'utilisateur a un certain potentiel d'apprendre plus rapidement les informations pertinentes qu'avec grep. L'inconvénient de la technique IR est qu'elle se base uniquement sur l'information lexicale dans le code source et ne considère pas l'information structurelle ;

- La technique de recherche de dépendances statiques est la technique la plus structurée des trois et elle peut être utilisée sans aucun outil de support. Les dépendances peuvent être suivies manuellement à travers le code. Par contre, elle dépend nécessairement de la compréhension correcte du programmeur des fonctionnalités des classes.

Dans l'analyse dynamique, il y a le problème d'exécuter le code, ce qui n'est pas toujours facile. Aussi l'inconvénient réside dans la génération des traces d'exécutions. Elles sont souvent nombreuses. Le choix des cas de tests ou des scénarios lors de l'exécution est un autre problème de ces techniques. Aussi, la limitation de telle analyse peut être la génération des traces d'exécutions pour traiter plusieurs fonctionnalités du système, alors qu'une demande de maintenance peut typiquement concerner une et une seule fonctionnalité particulière.

Pour les approches hybrides (statique et dynamique), elles ont leurs propres limitations [37, 47]. L'information statique est utilisée pour filtrer les traces d'exécutions. Comme cité auparavant, l'inconvénient réside dans la formulation des bonnes requêtes. Par suite, le programmeur doit posséder une bonne connaissance du système étudié.

Une comparaison des différentes approches pour la localisation des fonctionnalités dans les systèmes légataires a été présentée par Wilde [142]. Un résumé sur toutes les approches existantes peut être trouvé dans [20], alors qu'un résumé sur les outils industriels disponibles pour la localisation des fonctionnalités est présenté dans [133].

2.3 Conclusion

Le programmeur peut percevoir lors de la recherche que la fonctionnalité des classes est présentée de deux manières différentes. Une fonctionnalité composée (*composite functionality*) définie par toutes les classes qui la supportent. Une fonctionnalité locale

(*local functionality*) implémentée dans une classe et qui n'est pas déléguée à d'autres classes.

Dans les chapitres qui suivent, nous allons développer des techniques d'analyse de code afin d'identifier différentes manières d'implémenter les fonctionnalités. Il existe plusieurs types d'analyses qui peuvent être combinées entre elles [120]. Habituellement, ces techniques exigent une connaissance préalable du système par les utilisateurs afin de spécifier les parties du code source qui ont besoin d'être analysées.

Dans cette thèse, nous optons pour des approches qui se basent sur l'analyse statique pour l'identification des fonctionnalités dans du code orienté objets. Nous jugeons qu'une analyse statique a l'avantage d'être moins coûteuse au niveau de l'analyse du code et plus enrichissante vue les informations que nous pouvons extraire. Nous n'utiliserons pas l'analyse dynamique (traces d'exécution) vue que le programmeur doit avoir une connaissance préalable du système et vue la limitation du nombre de formulations des requêtes. Nous allons présenter une technique semi-automatisée pour l'identification des différentes fonctionnalités. Celle-ci n'exige pas une connaissance préalable du système par le développeur. Nous définissons une fonctionnalité par un ensemble de signatures de méthodes comme source d'informations.

La première approche effectue l'analyse du code afin d'identifier les fonctionnalités déjà existantes et reconnues par le concepteur. Nous proposons qu'une fonctionnalité déjà existante peut être héritée par une super classe ou en implémentant une interface. Elle peut aussi être déléguée par une autre classe à travers un agrégat. Ceci nous amène à identifier les différentes fonctionnalités existantes dans le code à travers l'héritage multiple ou la délégation. L'approche détaillée est présentée dans le chapitre 3.

La deuxième approche est basée sur l'identification des fonctionnalités non reconnues par le concepteur. Elles sont définies dans le code à chaque fois que le concepteur en a besoin. Dans ce cas, une fonctionnalité est caractérisée par un ensemble récurrent de méthodes dans le code. Cette récurrence reflète sa réutilisation. Notre contribution principale se manifeste par l'utilisation et l'application de la théorie de l'AFC (analyse formelle des concepts). Elle est détaillée dans le chapitre 5. Nous avons opté pour cette théorie puisqu'elle permet d'avoir des regroupements composés par des instances

de classes et leurs méthodes sous-jacentes, à l'inverse des méthodes statistiques qui se basent sur une analyse approximative où les regroupements peuvent être brouillés car ils ne sont pas explicitement définis (les regroupements sous forme d'ensembles ayant une intersection non vide). Pour valider chacune de nos approches, nous avons utilisé cinq outils libres. Des résultats prometteurs sont présentés dans le chapitre de validation.

CHAPITRE 3

PRINCIPES DE NOTRE APPROCHE

Notre approche vise à identifier les aspects fonctionnels ou fonctionnalités dans du code légataire orienté objets. Pour une application légataire orientée objets, il serait intéressant d'identifier et d'essayer d'isoler des fragments de code qui implémentent une exigence ou tout simplement une fonctionnalité particulière comme nous l'avons expliqué dans le chapitre précédent. Une telle identification et localisation permettent une meilleure compréhension, et une maintenance plus facile de l'application en cas de changement.

Certaines techniques telles le *refactoring* [58] ou les techniques orientées aspects [111], permettent d'isoler ces fonctionnalités et de les regrouper afin de pouvoir les composer en cas de besoin dans le programme. Par contre en l'absence des techniques orientées aspects, le développeur a besoin d'avoir recours à d'autres manières pour combiner les différentes fonctionnalités dans une même hiérarchie de classes. Les deux techniques les plus traditionnelles que l'on utilise pour intégrer, dans une classe, une fonctionnalité déjà reconnue en tant que telle est codée dans une classe, sont l'héritage multiple, quand le langage le permet, ou la composition/délégation. L'héritage multiple permet à une classe d'hériter différents comportements de plusieurs hiérarchies de classes. La délégation permet à une instance de classe de déléguer une tâche ou une fonction à une autre instance afin de l'accomplir.

Alternativement, lorsqu'un aspect fonctionnel n'a pas été reconnu par le développeur en tant que fonctionnalité indépendante et réutilisable, on en retrouve les éléments (méthodes et attributs) dans différents endroits de la hiérarchie, un peu à la manière de clonage de code. C'est ce que nous avons appelé "multiplication d'états". La notion de multiplication d'états est donc une manière ad-hoc où un développeur non expérimenté ajoute dans sa hiérarchie de classes, une fonctionnalité sous forme d'un patron d'attributs ou de méthodes dont la distribution est présentée d'une manière hiérarchique à chaque fois qu'il en a besoin. Il ne prend aucune mesure pour avoir une meilleure fac-

torisation ou conception. Comme illustré dans la figure 3.1, le même patron d'attributs définit par l'ensemble d'attributs *{foldingBackSeat, HasThirdRow}* se trouve dans deux endroits différents.

En principe, toutes ces techniques permettent l'appel et la réutilisation d'un comportement ou d'une fonctionnalité d'une ou de plusieurs classes dans un programme orienté objets. Dans ce qui suit, nous allons expliciter ces trois techniques afin de pouvoir ensuite développer les algorithmes pour identifier leur utilisation dans un code orienté objets légataire. Une analyse du résultat d'identification est nécessaire par l'utilisateur pour reconnaître si les éléments listés représentent réellement des aspects fonctionnels.

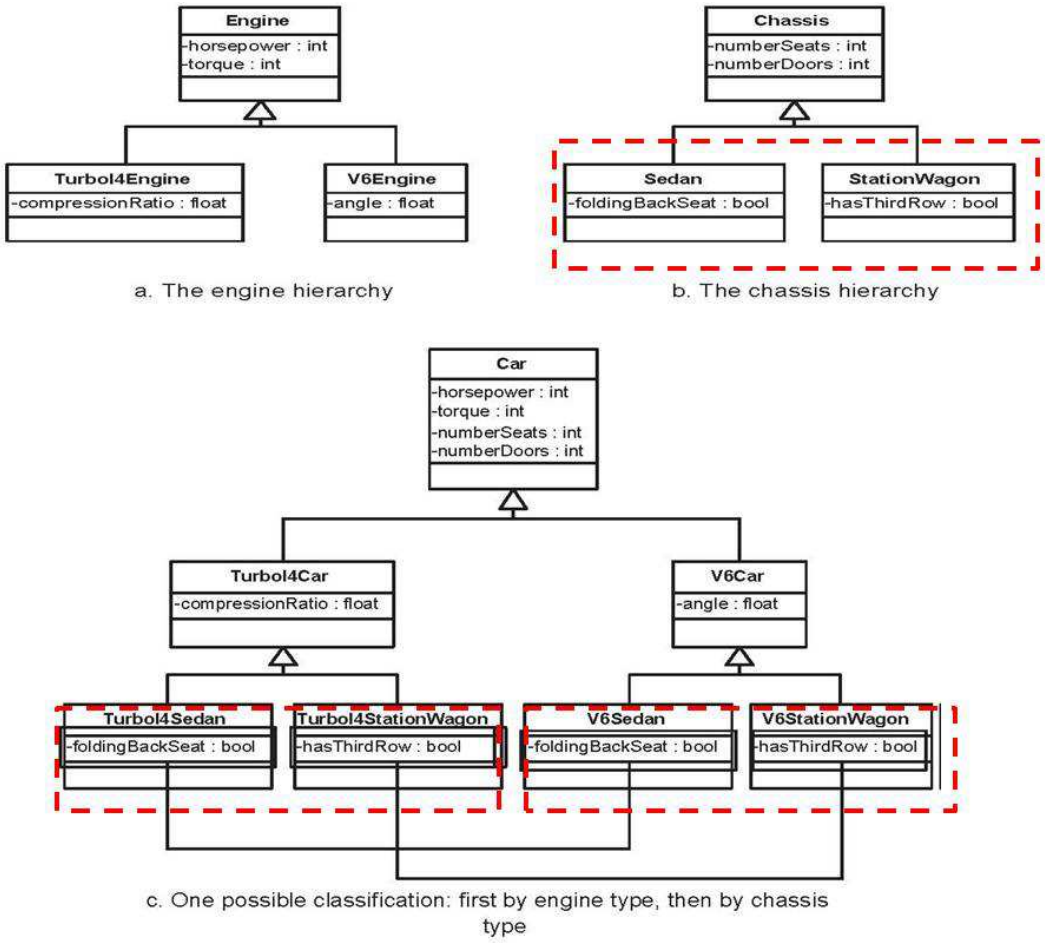


Figure 3.1 – Exemple de multiplication d'états

3.1 Approches pour l'identification des fonctionnalités

Nous définissons une fonctionnalité par un ensemble de méthodes. Nous faisons abstraction des attributs car ceux-ci sont généralement privés et donc accessibles via leur accesseurs qui sont pris en compte par nos algorithmes. Nous caractérisons un tel type de fonctionnalité par sa combinaison et sa réutilisation dans le code à travers différentes manières. Nous nous concentrons sur trois techniques qui permettent un tel ajout dans une conception orientée objets :

- L'héritage multiple : une fonctionnalité, qui est dans ce cas incarnée dans une classe, peut être héritée par une classe à travers l'héritage de classes issues de différentes hiérarchies de classes comme le montre la figure 3.2. Ainsi, une classe peut combiner plusieurs fonctionnalités issues de plusieurs hiérarchies.

- L'agrégation : une fonctionnalité aussi incarnée sous forme de classe, peut être réutilisée à travers cette fois-ci une délégation. Une classe peut appeler une fonctionnalité par l'intermédiaire d'un agrégat, auquel elle lui délègue l'implémentation de la fonctionnalité.

- La multiplication d'états : d'une manière ad-hoc, la fonctionnalité consistant en un ensemble d'éléments (méthodes) non structuré, est ajoutée directement dans les classes qui les utilisent à chaque fois que l'utilisateur en a besoin. La réutilisation consiste dans ce cas-ci à combiner le même patron de méthodes dans différents endroits.

Pour les applications orientées objets légataires d'une certaine complexité, il est fort probable que plusieurs classes combinent plusieurs aspects fonctionnels. En l'absence des techniques de programmation orientées aspects, les développeurs ont dû utiliser des techniques non-orientées aspects pour combiner les fonctionnalités, en l'occurrence, l'héritage multiple, l'agrégation, la multiplication d'états et possiblement d'autres. Comment faire alors pour 1) savoir si des classes dans un code légataire combinent plusieurs aspects, et 2) identifier ces aspects. Notre approche consiste en la démarche suivante :

- Identifier les différentes techniques qui peuvent être utilisées pour implémenter et composer différentes fonctionnalités dans une même hiérarchie de classes - nous avons identifié l'héritage multiple, la délégation et la multiplication d'états ;
- Pour chaque technique, caractériser son empreinte structurelle dans le code ;
- Développer une technique d'identification de cette empreinte dans le code ;
- Analyser le résultat obtenu (des classes potentielles ou des fragments de méthodes) pour vérifier s'ils représentent des aspects fonctionnels réutilisables.

Dans ce qui suit, nous allons présenter les différentes techniques citées et expliquer notre approche d'identification liée aux techniques d'héritage multiple et de délégation. Pour la multiplication d'états, elle sera définie dans la section 3.4 et explicitée dans le chapitre 5.

3.2 Fonctionnalités composées avec l'héritage multiple

3.2.1 Principe

La figure 3.2 illustre bien la situation de l'implémentation de fonctionnalités en utilisant l'héritage multiple. Chaque fonctionnalité est contenue dans une hiérarchie de classes. La classe *maClasse*, qui intègre plusieurs fonctionnalités, hérite des classes *E* et *C* de chaque hiérarchie. C'est le cas simple où les hiérarchies n'ont pas de racine commune, la classe hérite à partir des nœuds feuilles et devient elle aussi une classe feuille.

Cette situation est très simple à reconnaître : simplement identifier les instances de l'héritage multiple, c'est à dire identifier les classes dans la hiérarchie de classes qui ont plus qu'un ancêtre immédiat. Avec cette hypothèse, nous pouvons identifier les parties de la hiérarchie qui peuvent correspondre à des fonctionnalités par l'appel de méthodes de certains ancêtres.

La figure 3.3 montre un simple exemple qui illustre cette idée. La classe racine de la sous-hiérarchie commune aux deux hiérarchies, a deux super classes. Elle hérite les

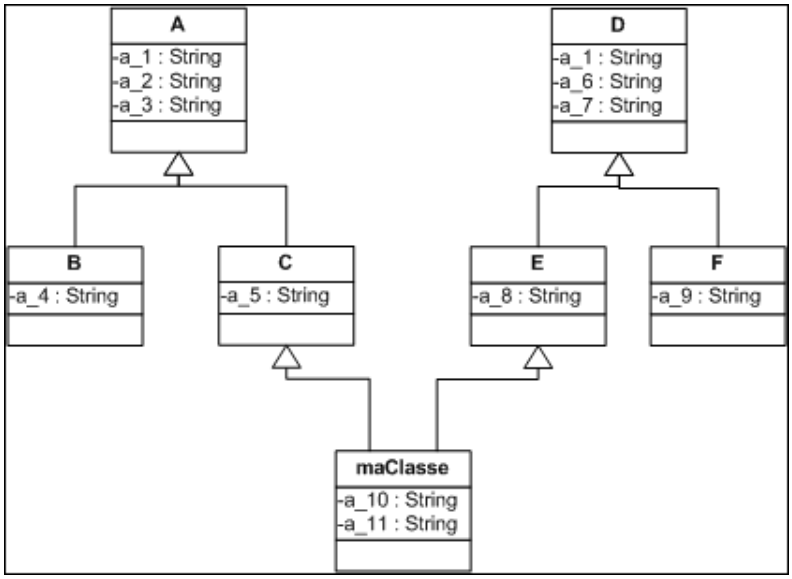


Figure 3.2 – L'implémentation de plusieurs fonctionnalités en utilisant l'héritage multiple

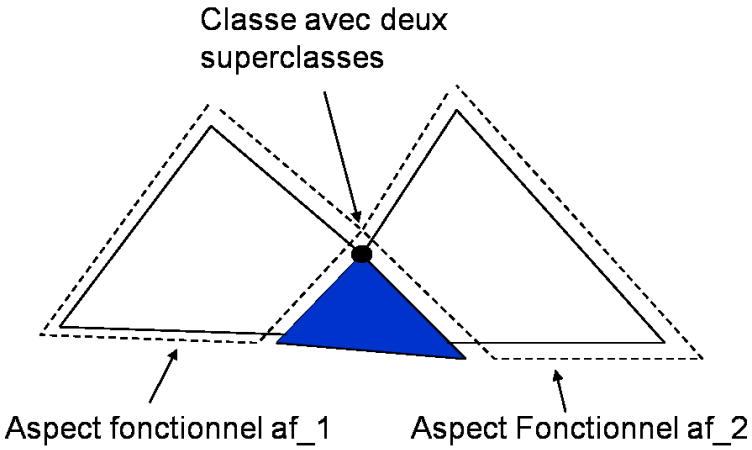


Figure 3.3 – Deux fonctionnalités ayant une sous-hiérarchie commune

méthodes de celles-ci, ce qui implique une réutilisation de ces méthodes. Dans le cas où cette classe racine est supprimée ainsi que sa sous hiérarchie, nous aurons deux hiérarchies complètement disjointes (au sens d'inclusion des classes) représentant chacune d'elle une fonctionnalité disjointe.

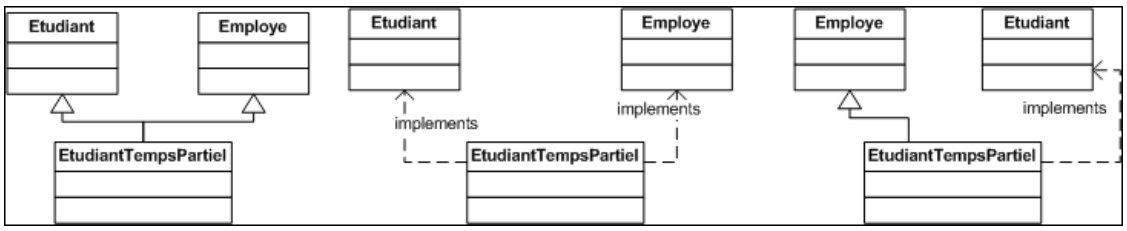


Figure 3.4 – Implémentaion de l'héritage multiple

La figure 3.4 présente les différents cas de l'héritage multiple dans un langage orienté objets. La classe *EtudiantTempsPartiel* peut hériter deux comportements différents celui d'un employé et celui d'un étudiant. Cette illustration se généralise au cas où la classe a plus de deux ancêtres, où ces ancêtres proviennent d'une ou de plusieurs hiérarchies distinctes. Dans la partie suivante, nous nous intéresserons à l'identification des instances de l'héritage multiple dans le cas des langages qui ne supportent pas la technique de l'héritage multiple. Dans le cas de Java par exemple, il offre juste les possibilités de l'héritage d'une classe et l'implémentation d'une ou plusieurs interfaces ou uniquement l'implémentation de plusieurs interfaces.

3.2.2 Cas de langages ne supportant pas l'héritage multiple

Peu de langages de programmation orientés objets supportent l'héritage multiple. Ce dernier peut provoquer un certain conflit d'appel quand, par exemple, deux ancêtres d'une même classe possèdent une méthode ayant la même signature. Comment différencier entre les deux méthodes lors d'un appel ? Ceci peut aussi augmenter la complexité du langage lors de son implémentation et une liaison dynamique qui devient plus coûteuse [90]. Ainsi, nous allons considérer les patrons de conception/codage pour simuler l'héritage multiple utilisés par les développeurs dans les langages qui ne le supportent pas.

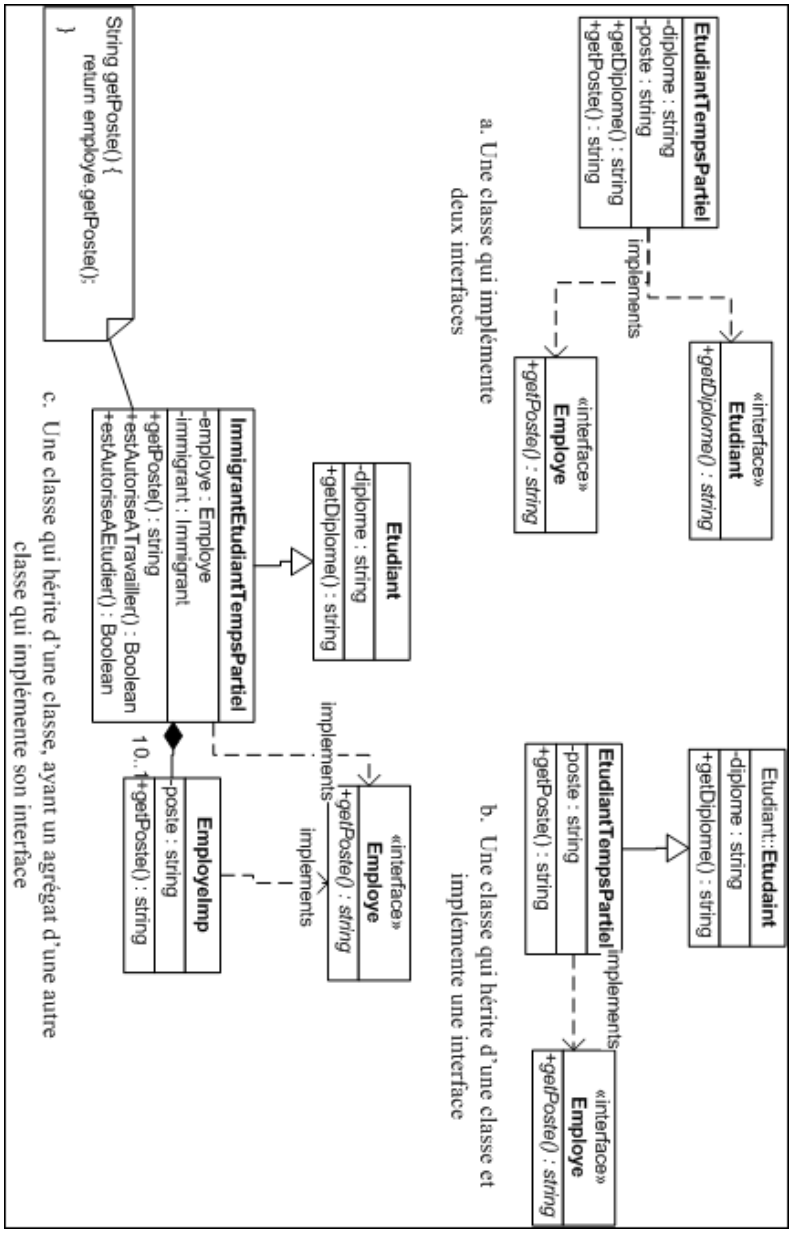


Figure 3.5 – Exemples de l'héritage multiple dans java

La figure 3.5 illustre comment un développeur Java peut simuler l'héritage multiple. On se sert de la relation «*implements*» entre classes et interfaces. En principe, les interfaces permettent d'ajouter un comportement nouveau différent de celui des membres d'une même famille de la classe. La figure 3.5.a présente le cas où une classe implémente deux interfaces. La classe **EtudiantTempsPartiel** implémente les interfaces **Etudiant** et **Employe**. Un étudiant à temps partiel peut être en même temps un employé et un étudiant. Dans le cas de la figure3.5.b, la classe hérite la première fonctionnalité **getDiplome()** de la classe **Etudiant** et implémente la deuxième fonctionnalité **getPoste()** issue de l'interface **Employe**. Dans ce cas, la classe utilise deux fonctionnalités en utilisant deux relations différentes. Dans le cas de la figure 3.5.c, la classe **EtudiantTempsPartiel** supporte les fonctionnalités de **Etudiant** et **Employe** à travers la combinaison de l'implémentation de l'héritage et de la délégation. Elle hérite la fonctionnalité **getDiplome()**, alors que pour la fonctionnalité **getPoste()**, elle la possède à travers un agrégat de type **Employe**. Ces exemples ne sont en aucun cas exhaustifs. Nous considérons le cas de deux aspects fonctionnels, mais nous pouvons avoir une classe qui implémente trois interfaces ou plus.

Par suite, pour le cas de la figure 3.5.a, il est clair que la classe **EtudiantTempsPartiel** implémente deux aspects fonctionnels, l'aspect *étudiant* et l'aspect *employé*. On peut aussi imaginer une situation où le concepteur définit, au préalable, une interface appelée **EtudiantEmploye** qui implémente les deux interfaces **Etudiant** et **Employe** et indique que la classe **EtudiantTempsPartiel** implémente cette interface. Ce qui signifie que pour identifier le cas de l'héritage multiple, il ne suffit pas de se restreindre des ancêtres immédiats d'une classe. Nous avons ainsi besoin de remonter dans la hiérarchie d'interfaces pour trouver les cas de l'héritage multiple. Finalement, notons que les cas des figures 3.5.a et 3.5.b représentent en quelque sorte une utilisation disciplinée d'interfaces. Par contre pour le cas de la figure 3.5.c, un concepteur novice peut ne pas se donner la peine de définir une interface pour **Employe**, ou si c'est le cas, de ne pas spécifier que la classe **EtudiantTempsPartiel** l'implémente. Il se contentera d'avoir l'héritage de la classe **Etudiant** et d'avoir un agrégat de type **Employe**. Ainsi pour la détection de l'héritage multiple, notre algorithme doit prendre en compte toutes ces pos-

sibilités.

3.3 Fonctionnalités composées par délégation à travers un agrégat

La délégation est une autre technique de conception qui peut être utilisée pour supporter plusieurs aspects fonctionnels. Elle est une technique de composition assez puissante pour la réutilisation, comme la technique d'héritage. Dans la délégation, deux objets sont impliqués pour la réalisation d'une demande ou d'une tâche : un objet récepteur qui délègue des opérations à un autre objet, son délégué [63]. Le terme "délégation" a une signification précise. La figure 3.6 montre de telles possibilités. En général, les objets ne réalisent pas tous les comportements dont ils ont besoin. Ils peuvent déléguer des comportements à d'autres objets existants afin de les effectuer comme étant des délégateurs. Nous présentons dans ce qui suit notre définition approximative des différents cas de délégation. Une classe A délègue une classe B ssi :

- Classe A définit un attribut (membre de données) de type B appelé b ;
- Classe A implémente le comportement de B, et
- L'implémentation dans A d'une méthode de B, transfère l'appel à l'attribut b.

Dans la figure 3.6.a, la classe **ImmigrantEtudiantTempsPartiel** délègue toutes les tâches à **EmployeImp** et **ImmigrantImp** :

- Il possède les attributs *immigrant* et *employe* de types **Employe** et **Immigrant** respectivement ;
- Il implémente les interfaces **Employe** et **Immigrant** ;
- L'implémentation des différentes méthodes des interfaces retournent les résultats de l'appel des méthodes correspondantes des attributs *employe* et *immigrant* respectivement. [1].

[1]Notons que cette situation ne correspond pas à la définition officielle de la délégation. L'attribut *employe*, par exemple, exécute *getPoste()* dans son propre contexte, à l'opposé du contexte de l'expédition des instances **ImmigrantEtudiantTempsPartiel**, comme le préconiserait la délégation

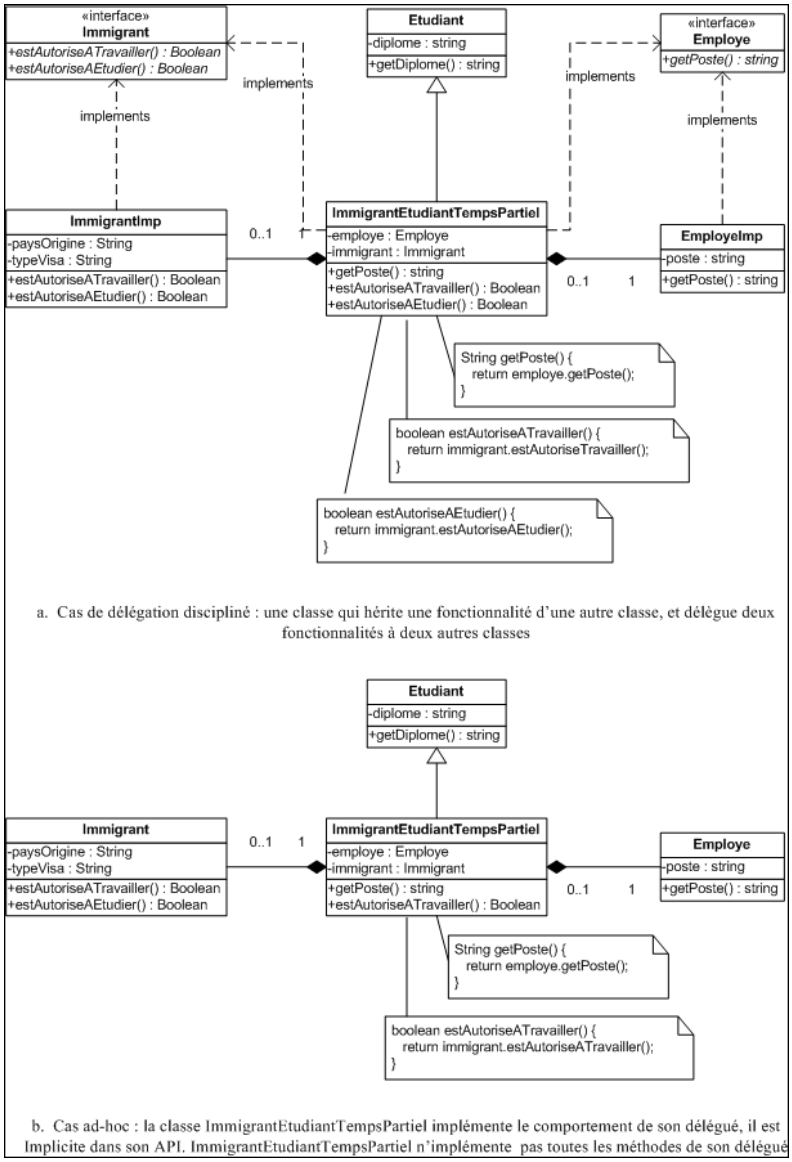


Figure 3.6 – Différents aspects fonctionnels supportés par délégation : deux patrons différents

Par contre, la figure 3.6.b montre une utilisation "indisciplinée" de délégation. Dans ce cas, le fait que **ImmigrantEtudiantTempsPartiel** implémente le comportement de **Employe** ou **Immigrant** est seulement implicite dans son API, à l'opposé d'être explicite à travers l'implémentation d'une interface commune. Pour notre proposition, cela signifie que nous avons besoin d'examiner l'ensemble de méthodes implémentées par chaque classe pour s'assurer que **ImmigrantEtudiantTempsPartiel** implémente le comportement de ses déléguées. En pratique, il est peu probable que toutes les méthodes des classes auxquelles on délègue les tâches (classes **Employe** et **Immigrant** dans notre cas) soient implémentées par celui qui délègue- **ImmigrantEtudiantTempsPartiel**. La figure 3.6.b montre que la classe **ImmigrantEtudiantTempsPartiel** implémente / délègue seulement une méthode de **Immigrant**, nommée *estAutoriseATravailler()* dans notre exemple. La figure 3.6.b représente un cas ad-hoc de délégation, un cas plus faible qu'une délégation disciplinée telle illustrée par la figure 3.6.a. Par conséquent, l'algorithme d'identification des instances de délégation dans un code légataire doit aussi considérer les cas d'une implémentation partielle des méthodes. Dans ce cas, il n'est pas obligatoire d'avoir une implémentation de toutes les méthodes de la classe à laquelle on confie la réalisation d'une fonctionnalité spécifique. Nous pourrions utiliser un pourcentage dans les conditions de notre définition de délégation :

- Classe A a une variable d'instance du type B et
- Classe A implémente au moins un pourcentage α du comportement de B.

Cependant, une telle heuristique peut produire plusieurs faux-positifs et plusieurs faux-négatifs pour deux raisons. Premièrement, les classes du domaine vont typiquement avoir peu de méthodes du domaine et un ensemble de méthodes utilitaires telles les constructeurs, les accesseurs, les convertisseurs, les hashers, les comparateurs, etc. De plus, le critère du pourcentage seul n'est pas un indicatif. Il dépendra des méthodes déléguées. Deuxièmement, la classe qui délègue a besoin seulement de réutiliser / déléguer un aspect- lire une méthode- pour faire une instance de délégation légitime.

Comme pour l'héritage multiple, la légitimité de la délégation dépend sur "ce qui est délégué à". Dans la figure 3.6, le délégateur et les délégués sont clairement les classes du

domaine. Nous devrions plutôt dire appartiennent au même domaine : en effet, si notre domaine d'application est l'IUG (Interfaces Utilisateurs Graphiques), les délégués qui font partie des packages graphiques de Java (java.awt.* ou javax.swing.*) représentent des instances légitimes de délégation. Nous devrions distinguer ces instances de celles où les classes du domaine délèguent aux classes utilitaires. Un exemple typique est quand une classe utilise une collection pour se référer aux objets associés : on ajouterait typiquement des méthodes déléguées pour énumérer les éléments de la collection, ou pour ajouter un élément à la collection. L'extrait du code suivant illustre ce cas :

```
class Personnel {
    private Collection members;
...
Iterator members() {
    return members.iterator();
}
...
public Object add(Employee emp) {
    return members.add(emp);
}
...
}
```

Dans ce cas, nous ne pouvons pas dire que la classe **Personnel** délègue à l'interface/classe **Collection**. Ce code illustre un autre problème potentiel qui peut arriver avec des instances légitimes de délégation : un développeur peut *renommer* une méthode déléguée dans un délégateur. Dans notre exemple, la méthode *iterator()* de **Collection** a été renommée *members()* dans la classe **Personnel**. Nous devons donc relaxer notre définition de délégation de méthodes : nous devrions peut être ignorer les noms de méthodes, et nous intéresser à leurs types de retour et à leurs types des paramètres.

Un algorithme pour la détection des cas de délégation doit prendre en considération toutes ces variations. Comme pour le cas de l'héritage multiple, l'algorithme doit utiliser une combinaison d'heuristiques avec certaines connaissances du domaine d'application et de l'API de Java. Encore une fois, seule l'expérimentation va nous aider à définir et à raffiner ces heuristiques.

Dans ce qui suit, nous allons définir une fonctionnalité présentée à travers la notion de multiplication d'états. Nous détaillerons cette notion à travers des exemples et nous élaborons les caractéristiques de ce cas-ci.

3.4 Fonctionnalités composées par multiplication d'états

L'idée de base de notre approche est qu'une fonctionnalité peut être représentée de deux manières différentes dépendamment si la fonctionnalité est connue à l'avance ou non. La première manière exige que les membres contribuant à une fonctionnalité soient connus à l'avance et par suite ils peuvent être factorisés à travers une classe racine ou une interface (la fonctionnalité est acquis à travers l'héritage ou par implémentation d'interface comme indiqué dans la figure 3.4). La fonctionnalité peut aussi être déjà implémentée dans l'application, et dans ce cas elle peut être appelée à travers un agrégat par délégation partout où le développeur a besoin. Par contre une autre façon de conception, la fonctionnalité n'est pas connue à l'avance et donc elle est représentée sous une forme étendue : les membres contribuant sont définis à chaque endroit de la hiérarchie de classes où la fonctionnalité est exigée. La forme étendue peut être sous une forme hiérarchique pour la distribution des membres. Par conséquent, l'ensemble de membres de méthodes ou de données exprimant une fonctionnalité est redéfini et est combiné dans *plusieurs endroits*.

Nous référons par l'*implémentation ad-hoc* les cas où le développeur ne prend aucune mesure spéciale pour séparer les artefacts implémentant des fonctionnalités particulières par rapport aux autres hiérarchies de classes. Comment alors, reconnaître qu'un ensemble de membres de classes (données et méthodes) contribue à une fonctionnalité ? Des résultats prometteurs par l'analyse de code (*slicing*) ont été effectués dans un travail de Dagenais-Mili [38] en identifiant les classes et les membres de données qui contribuent à une seule fonctionnalité.

Dans une mise en œuvre ad-hoc, considérons un cas spécial - que nous appelons *la multiplication d'états*. La notion de *multiplication d'états* est la situation qui comprend plusieurs fonctionnalités combinées. Un modèle illustrant ce cas est présenté par la fi-

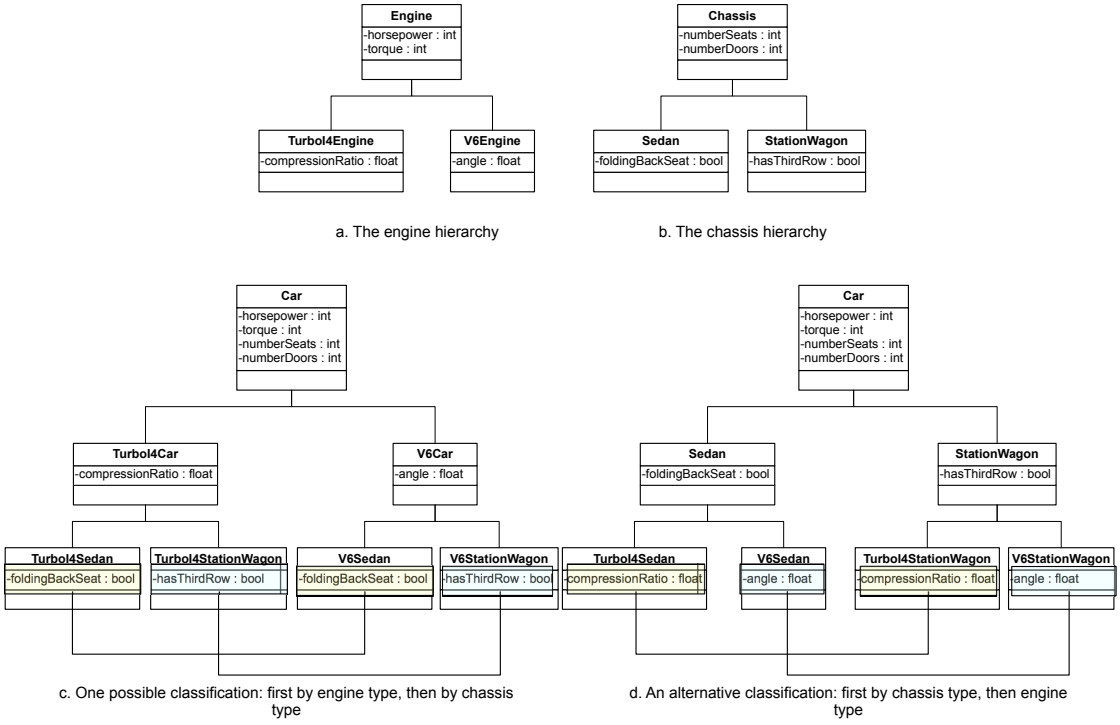


Figure 3.7 – Exemples supportant plusieurs fonctionnalités dans le cas de multiplication d'états

gure 3.7. Par exemple, les voitures peuvent avoir plusieurs formes de châssis, y compris des berlines, des coupées, des familiales, etc. Elles peuvent aussi avoir divers types de moteurs (moteur à 4 cylindres ou à 6 cylindres). Les figures 3.7.a et 3.7.b montrent des hiérarchies de châssis et des types de moteurs, qui pourraient se combiner en une seule hiérarchie de classes. Ainsi, une voiture donnée aura une combinaison des deux fonctionnalités en ayant naturellement un type de chassis (la catégorie de la voiture) et un type de moteur associé. Les figures 3.7.c et 3.7.d montrent deux classifications possibles de voiture. Ainsi, pour clarifier notre utilisation de la notion "multiplication d'états", considérons la classe *Voiture*. Dans les deux figures 3.7.c et 3.7.d, chaque voiture combine "des variables d'état" des châssis et des composants de moteur, c-à-d les deux classifications possibles de voiture sont présentes. Les hiérarchies dans les figures 3.7.c et 3.7.d correspondent aux situations typiques qui justifieraient l'utilisation du patron *stratégie*. Ce patron, en fait, cherche principalement à séparer un objet de ses comportements en encapsulant ces derniers dans des classes à part. Il est utilisé lorsqu'un objet peut avoir plusieurs comportements différents [89]. En effet, le patron *stratégie* traite ces types de situations par (voir figure 3.8) :

- la combinaison des fonctionnalités sont dans leurs propres hiérarchies,
- et laisser l'objet de base (*Voiture*) se référer aux fonctionnalités (le type de châssis et le type de moteur) par l'agrégation / la délégation [64] !

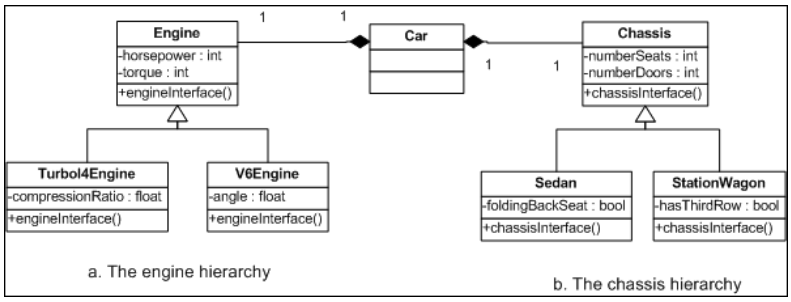


Figure 3.8 – Exemple de patron stratégie

La question qui se pose si telle conception avec la patron *stratégie* est plausible ? Très probablement si les hiérarchies de type *Moteur* et *Châssis* sont présentées à l'avance à un

développeur, il essayera de les combiner selon ses besoins en utilisant l'héritage multiple ou la délégation présentée dans les sections 3.2 et 3.3. Par contre, les hiérarchies comme celles dans les figures 3.7.c et 3.7.d pourraient bien résulter d'une conception qui ajoute les fonctionnalités point par point, c'est-à-dire débutant avec la première fonctionnalité du domaine de problème (par exemple dans le cas de la figure 3.7.c, débuter par définir la hiérarchies des types de moteurs d'une voiture). Ensuite avoir le besoin d'ajouter la deuxième fonctionnalité sous forme de spécialisation des nœuds des feuilles de la hiérarchie de classes (pour chaque type de moteur, définir le type de chassis).

Une fonctionnalité est reconnue par l'occurrence d'un *patron structurel* de données et des méthodes dans différents endroits de la hiérarchie de classes. Nous remarquons qu'il inculque un comportement particulier sur les classes auxquelles il s'attache.

En résumé, nous formulons l'hypothèse qu'un patron récurrent de données et de fonctions qui apparaisse à différents endroits d'un code légataire, reflète probablement une fonctionnalité distincte, que le développeur n'a pas encore reconnue en tant que telle pour la représenter sous forme de classe ou interface à part. Cette fonctionnalité n'a pas été identifiée par le développeur à l'avance. De tels patrons récurrents peuvent être retrouvés en utilisant des techniques de factorisation. Une analogie pratique est la notion de *clonage ou multiplication de code* où les mêmes fragments de code - modulo le changement de nom de variables ou d'autres différences syntaxiques mineures - peuvent être retrouvés dans des branches différentes de la hiérarchie des classes [50]. Dans un chapitre ultérieur, nous allons détailler cette notion et décrire les différentes approches et démarches que nous avons suivies afin d'aboutir à un algorithme d'identification de ce cas-ci.

3.5 Conclusion

Dans ce chapitre, nous avons explicité les différentes techniques qui permettent de combiner et réutiliser différentes fonctionnalités dans un code orienté objets légataire en l'absence des approches orientées aspects. Nous avons ciblé trois techniques à savoir l'héritage multiple, la délégation à travers un agrégat et la multiplication d'états. L'idée

était d'étudier comment chacune d'elle permet l'utilisation d'une fonctionnalité donnée afin de développer des algorithmes d'identification pour reconnaître les différentes fonctionnalités dans le code.

Dans le chapitre qui suit, nous présentons les algorithmes d'identification et de localisation des différentes fonctionnalités dans le code. Nous présentons une fonctionnalité comme étant un ensemble de méthodes qui peut être issue de l'héritage multiple ou appelé par délégation à travers un agrégat. Concernant la notion de multiplication d'états, elle est représentée par un patron récurrent de données ou de méthodes. L'approche d'identification ainsi que l'algorithme seront présentés et détaillés dans le chapitre 6.

CHAPITRE 4

ALGORITHMES D'IDENTIFICATION DES FONCTIONNALITÉS LIÉS PAR HÉRITAGE MULTIPLE ET PAR DÉLÉGATION

Dans l'absence des techniques de programmation orientées aspects, les développeurs doivent recourir à la conception et à la programmation orientée objets traditionnelles pour implémenter les différentes fonctionnalités (features). Suite au chapitre précédent, ce chapitre décrit comment nous pouvons identifier la combinaison et la réutilisation des différentes fonctionnalités existantes grâce aux techniques de conception orientés objets citées auparavant. Les algorithmes d'identification sont générés à partir de ces techniques [49]. Nous identifions trois techniques de conception orientées objets [49] :

- Héritage multiple,
- Délégation à travers l'agrégation,
- Multiplication d'états.

Dans ce qui suit, nous discutons comment nous pouvons identifier les instances des deux premières techniques, l'héritage multiple et la délégation par agrégation. En suite, nous présentons les algorithmes pour reconnaître leurs instances dans le code. Concernant la multiplication d'états, nous allons la décrire et détailler la démarche suivie pour aboutir à l'algorithme d'identification dans le chapitre qui suit.

4.1 Description et algorithme d'identification des instances de l'héritage multiple dans des programmes Java

4.1.1 Description des cas de l'héritage multiple

L'héritage multiple n'est pas supporté par tous les langages de programmation orientée objets. Par exemple, le langage Java ne supporte pas l'héritage multiple. Alors, l'idée

est d'étudier les différentes manières d'implémenter l'héritage multiple dans un tel langage. Nous allons explorer les différentes possibilités qu'un programme Java semble utiliser l'héritage multiple. Il en existe plusieurs, par exemple Java distingue entre les types (et sous-types) et l'héritage. Nous nous concentrons sur la combinaison de plusieurs interfaces et la combinaison d'interfaces et de classes. Les différentes possibilités d'implémentation de fonctionnalités ne sont en aucun cas exhaustifs. Nous considérons le cas de deux aspects fonctionnels : nous pouvons avoir une classe qui implémente trois interfaces ou plus.

4.1.2 Algorithme d'identification des instances de l'héritage multiple dans des programmes Java

Pour filtrer et identifier les fonctionnalités d'un programme, nous gardons uniquement les classes et les interfaces propres au programme. Les interfaces qui font parties de l'API Java sont à ignorer. Les interfaces qui ne reflètent pas une fonctionnalité telles les interfaces qui peuvent représenter simplement des constantes ou des interfaces appelées interfaces marqueurs doivent aussi être ignorées.

Un algorithme d'identification de l'héritage multiple peut identifier certaines situations qui ne représentent pas des cas d'aspects fonctionnels multiples. Certaines fonctionnalités ne seront pas pris en compte car elles illustrent des aspects infrastructurels ou utilitaires. Souvent, les interfaces sont utilisées pour ajouter des fonctionnalités aux objets du domaine qui supportent un "contrat de service" comme le cas d'un aspect infrastructurel. Par exemple, dans l'architecture EJB 2.x (Enterprise JavaBeans), une classe de session bean a dû mettre en œuvre l'interface ***SessionBean*** pour bénéficier des services fournis par le hôte "J2EE container". D'une manière similaire, pour être capable de comparer les objets ou les trier, les classes doivent mettre en œuvre l'interface ***Comparable***, laquelle possède une seule méthode. Il existe aussi les interfaces utilisées pour représenter les constantes et les interfaces marqueurs qui n'incluent aucune méthode mais qui marquent les classes qui sont traitées par d'autres API Java.

Ceci soulève la question sur comment nous pourrons distinguer tels cas à partir des interfaces qui représentent des aspects fonctionnels légitimes. En fait, nous aurons besoin

de compter sur une combinaison de nos connaissances des API Java et quelques heuristiques. Par exemple, si nous analysons une application MIS (Management Information System), et nous essayons d'identifier les aspects fonctionnels, nous pouvons ignorer sans risque toutes les interfaces qui font parties de l'API Java de base (comme packages java.*, javax.*) ainsi que les classes utilitaires (comme org.omg.*, org.w3c.*). La liste des APIs à exclure dépend des applications à analyser. Par exemple, si nous analysons un package de structures graphiques, les interfaces des packages AWT (java.awt.*) et Swing (javax.swing.*) sont pertinentes au domaine. Le jugement du développeur sur la pertinence de la réutilisation d'API est nécessaire. Des heuristiques additionnelles doivent considérer le type d'interfaces. Par exemple, les interfaces marqueurs et les interfaces constituées juste des constantes doivent être ignorées. Ainsi, notre algorithme doit prendre en considération tous ces cas pour la détection de l'héritage multiple.

L'algorithme d'identification des classes avec l'héritage multiple est comme suit :

Input: ensemble $\mathscr{C}$ des classes dans un programme P
Output: Liste de classes candidates implémentant plusieurs fonctionnalités par héritage multiple *CandClassesList*
while $\mathscr{C} \neq \emptyset$ **do**
  foreach *classe* $c \in \mathscr{C}$ **do**
    SUPER(c) $\longleftarrow$ {toutes les Superclasses(c) }$\bigcup$\{ les interfaces implémentées directement par c\}
    Eliminer de SUPER(c) toutes les interfaces de l'API Java non pertinentes au domaine de l'application
    Eliminer de SUPER(c) toutes les interfaces marqueurs
    Eliminer de SUPER(c) toutes les interfaces constantes
    if $|SUPER(c)| \geq 2$ **then**
      add(c, *CandClassesList*)
    end
  end
end

Comme illustré dans l'exemple de la figure 4.1, notons que notre algorithme peut

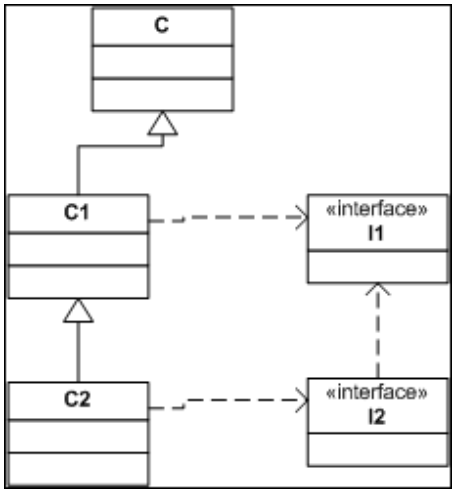


Figure 4.1 – Exemple de l'héritage multiple

retourner des paires de classes candidates C_1 et C_2 tel que SUPER(C_1) = $\{C,\ I_1\}$, pour une interface I_1, et SUPER(C_2) $=\ \{C_1, C,\ I_2\}$, où I_2 est un sous-type de I_1. En d'autres mots, C_2 spécialise C_1, et implémente une spécialisation de l'interface implémentée par C_1. Pour notre algorithme, ces cas ne sont pas considérés comme des cas séparés. Lors des propositions d'identification de fonctionnalités potentielles, la superclasse (C_1) et l'interface I_1 sont présentés comme étant elles représentent deux fonctionnalités différentes. L'algorithme est de $O(n)$, avec n est le nombre de classes.

Dans la figure 3.4, les fonctionnalités (*employe* et *etudiant*) peuvent être illustrées par l'implémentation de deux interfaces ou l'implémentation d'une interface et la spécialisation d'une super classe. À travers notre algorithme d'identification d'instances d'héritage multiple, comme les classes "*Etudiant*" et "*Employe*" ne sont ni des classes marqueurs, ni des classes constantes et ne sont pas des classes issues de l'API Java, la classe "EtudiantTempsPartiel" sera candidate à être une classe qui combine ces deux fonctionnalités.

Après exécution de l'algorithme d'identification, une liste des candidats est présentée à l'utilisateur afin de les analyser. Le jugement du développeur est toujours nécessaire pour trancher si tel candidat combine et réutilise formellement les fonctionnalités proposées.

4.2 Description et algorithme d'identification des instances d'agrégation dans des programmes Java

Une autre manière de combiner et de réutiliser des fonctionnalités déjà existantes peut être implémentée à travers une relation de délégation. C'est une composition d'objets. L'algorithme d'identification de ce type de relation se focalise sur les méthodes implémentées par les classes et vérifier si elles ont les mêmes signatures que les méthodes implémentées par leurs attributs.

La description de notre algorithme est la suivante :

Input: ensemble $\mathscr{C}$ des classes dans un programme P

Output: Liste de classes candidates implémentant plusieurs fonctionnalités par délégation *CandClassesList*

while $\mathscr{C} \neq \emptyset$ **do**
  foreach *classe* $c \in \mathscr{C}$ **do**
    DOMINT(c) $\longleftarrow$ { signature(m) / m méthode de c }
    foreach *attribut at de c de type T avec T non dans API Java* **do**
      Calculer $DOMINT(T)$
      if $DOMINT(C) \bigcap DOMINT(T) \neq \emptyset$ **then**
        foreach *signature(m)* $\in DOMINT(c) \bigcap DOMINT(T)$ **do**
          Les méthodes m correspondantes de c délèguent à d'autres méthodes de T Marquer classe c comme implémentation potentielle de $DOMINT(at)$
          add(c, *CandClassesList*)
        end
      end
    end
  end
end

DOMINT(C) représente l'interface du domaine de la classe C. Les méthodes héritées de l'API Java comme par exemple les méthodes *clone(), hash(), toString()*, etc sont

ignorées et supprimées. DOMINT(T) pour un attribut *at* est l'interface du domaine du type *at*. La signature d'une méthode *m* est définie par la paire composée des paramètres d'entrées et du paramètre de sortie $((Inp_1, Inp_2, ...Inp_n), Out)$ où Inp_i est le type du ième paramètre d'entrée de la méthode *m*, et *Out* est le type du paramètre de sortie. Les signatures de méthodes considérées sont celles ayant la même signature à la délégation (voir figure 4.2). S'il existe une seule méthode d'une classe C qui délègue une partie de son traitement à une méthode parmi un de ses attributs, la classe C est marquée comme une implémentation potentielle de l'interface de cet attribut. Le nombre de méthodes déléguées n'est pas considéré comme un indicateur pour la délégation. La liste des candidats est filtrée manuellement en inspectant minutieusement les résultats. L'algorithme est de $O(n * p)$, avec n est le nombre de classes et p est le nombre d'attributs maximal dans une classe.

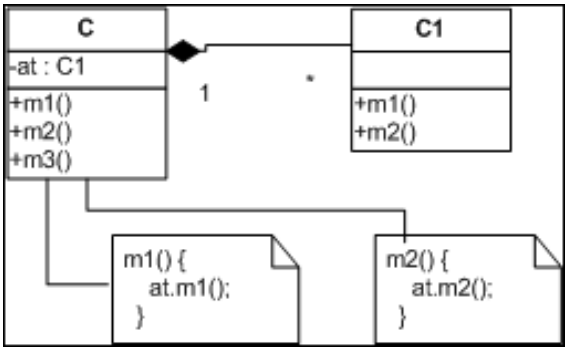


Figure 4.2 – Exemple de délégation

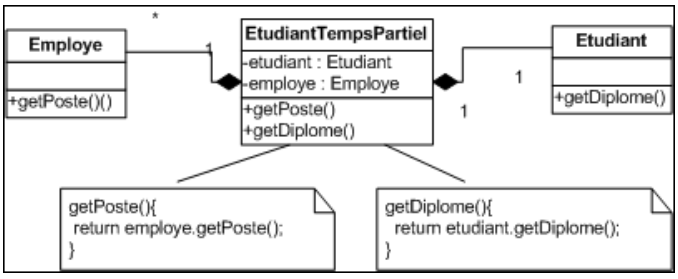


Figure 4.3 – Exemple d'implémentation de délégation dans java

En appliquant l'algorithme sur un exemple standard, voir la figure 4.3, l'algorithme

identifie que la classe "*EtudiantTempsPartiel*" est une classe candidate par délégation. Elle possède deux agrégats de type "*Employé*" et "*Étudiant*". Après analyse de ce cas, nous jugeons que telle classe combine les fonctionnalités des deux classes. La réutilisation se manifeste par l'appel des méthodes telles *getPoste()* et *getDiplome()* de chacune des classes déléguées. Le jugement de l'utilisateur pour interpréter les résultats est toujours nécessaire afin d'identifier les différentes fonctionnalités.

4.3 Conclusion

Dans ce chapitre, nous avons explicité les algorithmes d'identification des fonctionnalités liées par l'héritage multiple et par la délégation à travers un agrégat. Nous avons expliqué l'application des algorithmes d'identification à travers des exemples simples illustrant bien ces cas. Il reste des points à améliorer et à considérer, par exemple dans le cas de l'algorithme d'identification par délégation, où nous pourrions introduire l'identification à travers non seulement l'agrégat mais aussi à travers les paramètres ou les attributs de la méthode. C'est un point à considérer dans les travaux futurs.

Pour la notion de multiplication d'états, nous avons pensé à utiliser l'analyse formelle de concepts (AFC) [65] parce qu'elle représente une récurrence de patron de données ou de membres dans un code. Nous explorerons l'utilisation de l'AFC afin d'identifier les patrons récurrents dans des applications légataires orientées objets. L'AFC a été reconnue particulièrement appropriée pour l'analyse, la restructuration et la refactorisation d'applications orientées objets [15, 68, 88, 116, 132, 135, 136]. Elle permet des regroupements conceptuels de collection d'objets avec leurs propriétés communes. À cette fin, nous allons présenter dans le chapitre suivant cette théorie, l'utilisation des algorithmes de décomposition pour l'identification suivie de différentes preuves qui montrent la non possibilité d'utiliser ces algorithmes pour identifier ce format de patrons récurrents dans le code.

CHAPITRE 5

MÉTHODES MATHÉMATIQUES POUR FOUILLES DE RÉGULARITÉ

La notion de *multiplication d'états* illustre la situation où une fonctionnalité se retrouve à plusieurs endroits dans le code. Comme nous l'avons déjà définie dans le chapitre 3 section 3.4, nous rappelons qu'une fonctionnalité reflète un rôle spécifique qui est représentée par un ensemble d'attributs ou de méthodes. Elle est identifiée par l'occurrence d'un *patron structurel* de données et des méthodes dans différents endroits dans la hiérarchie de classes. Un tel patron assure un comportement particulier. La figure 5.1 est une hiérarchie de classes qui présente un exemple de *multiplication d'états*. Il s'agit de la fonctionnalité illustrée par l'ensemble d'attributs $\{capabilities, schedule, assemblyLine, licenceClass\}$ qui se retrouve dans deux endroits différents.

En général, la multiplication d'états est une anomalie de conception qui résulte de l'inexpérience ou de l'inattention du concepteur qui crée des combinaisons entre différentes fonctionnalités. À chaque endroit de la hiérarchie où la combinaison est nécessaire, il combine des caractéristiques[1] provenant de deux fonctionnalités ou plus. Cela est fait de manière plus ou moins exhaustive (dépendamment des combinaisons désirées).

Par ailleurs, la combinaison des fonctionnalités peut avoir l'allure d'un produit entre hiérarchies : sur un canevas constitué de la hiérarchie d'une des fonctionnalités, des copies d'une deuxième hiérarchie sont alors insérées (voir figure 3.7). Dans d'autres cas, les copies pourront reprendre qu'une partie de la deuxième hiérarchie incluant uniquement les classes pertinentes pour le concepteur à un endroit bien précis de la première hiérarchie. La combinaison de fonctionnalités est présente à chaque fois que le concepteur a besoin et parsuite elle peut être présente à plusieurs endroits dans la hiérarchie.

Notre but dans la partie qui suit, est de concevoir des approches pour l'identification de ces parties de hiérarchies situées dans plusieurs endroits. Notre hypothèse est que dans une hiérarchie de classes globale, il peut exister des hiérarchies facteurs. Une

[1] attributs ou méthodes

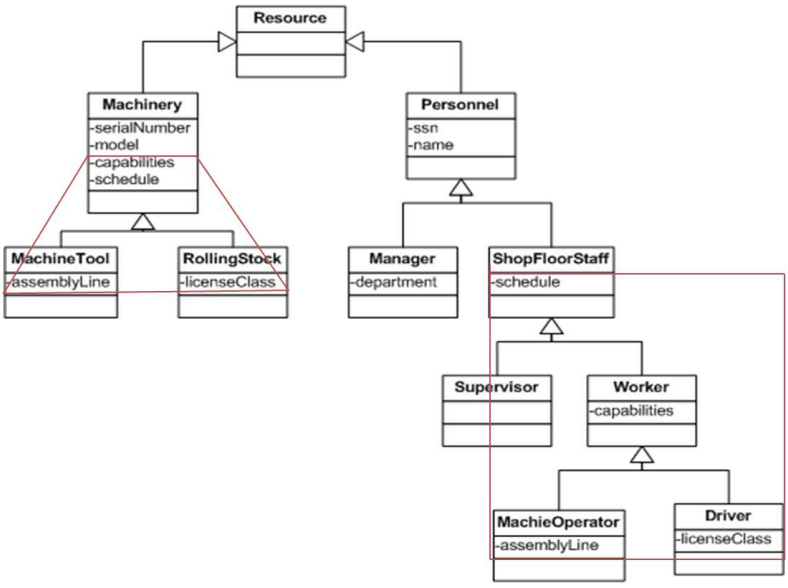


Figure 5.1 – Exemple d'une fonctionnalité récurrente

récurrence peut refléter une hiérarchie facteur. Le cas de multiplication d'états peut illustrer une représentation de hiérarchie plus stricte avec plus de contraintes sur la manière de combiner les facteurs (comme présenté dans l'exemple de la figure 3.7). Dans certains cas, ces parties peuvent être un ensembles de membres récurrents dans une hiérarchie de classes sans nécessairement avoir la structure d'une hiérarchie.

Nous allons présenter dans ce chapitre différentes approches que nous allons explorer afin d'identifier les fonctionnalités récurrentes existantes dans un code orienté objets.

5.1 Les représentations graphiques et la fouille de motifs de graphe

Une approche préliminaire consiste à appliquer des techniques de fouille de données pour détecter chacune des copies individuelle en tant qu'occurrence d'un patron structurel unique. Nous considérons la notion de graphe vu notre besoin d'identifier les occurrences multiples dans du code orienté objets. Or, la notion de graphe peut nous être utile pour retrouver les occurrences sur la base uniquement de la structure et non sur la sémantique. En effet, la hiérarchie pourrait être vue comme un graphe étiqueté où les classes sont des nœuds et les agrégations/liens de spécialisation sont des arcs.

De plus, les nœuds porteraient des étiquettes multiples correspondant à leurs membres déclarés. La figure 5.2 montre la représentation d'une hiérarchie de classes sous forme d'un graphe étiqueté orienté. Chaque classe est représentée par un nœud. Chaque nœud contient les membres associés à la classe. Les relations d'héritage sont représentées par des arcs étiquetés par "*is-a*". L'idée est d'identifier les occurrences d'un même ensemble de méthodes.

Les algorithmes de fouilles de données par les graphes cherchent à identifier les patrons de même structure. Ils sont utilisés pour identifier les patrons isomorphes. Or généralement, il existe des cas de hiérarchies similaires de point de vue sémantique, mais ils ont des structures différentes. Une hiérarchie récurrente n'a pas toujours la même structure dans les différents endroits où elle se trouve. Notre exemple de la figure 5.1 illustre bien ce cas. La fonctionnalité illustrée par l'ensemble des attributs $\{capabilities, schedule, assemblyLine, licenseClass\}$ est implémentée dans deux sous-hiérarchies d'attributs distinctes. La distribution des attributs est la même sauf pour le cas de l'attribut *{capabilities}*. Dans la sous-hiérarchie de droite, cet attribut se situe dans la sous-classe *Worker*. Dans les deux cas, bien que la répartition des attributs n'est pas la même, la sémantique est identique. Par suite, les deux sous-hiérarchies représentent deux patrons équivalents du point de vue sémantique, mais du point de vue structure ils sont différents. L'utilisation de tels algorithmes suppose que les copies recherchées doivent avoir la même structure et les étiquettes soient réparties de manière exactement identique au sein des graphes des copies. Un autre handicap important des approches par fouille par graphes est que les représentations en tant que graphes ignorent la sémantique spécifique des arcs, en particulier celle des liens de spécialisation.

Nous en déduisons que la structure de la hiérarchie ne peut être considérée comme un critère d'identification. Ainsi, la notion de graphe ne peut être utilisée puisque la structure ne peut être un guide pour notre identification.

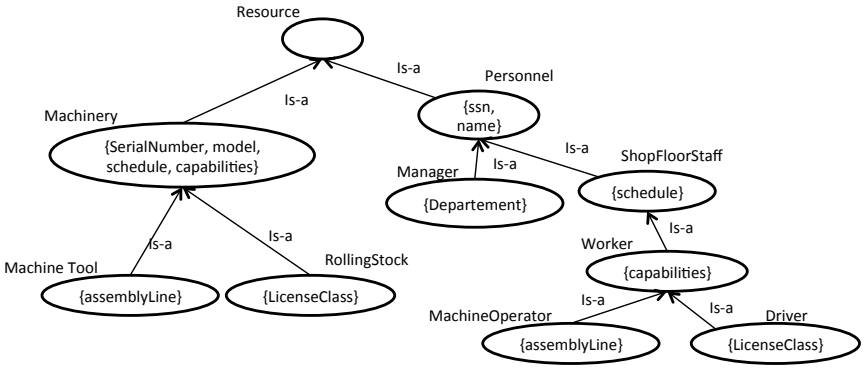


Figure 5.2 – Représentation de la hiérarchie de classes de la figure 5.1 sous forme d'un graphe

5.2 Les treillis de concepts : une alternative aux graphes

Afin d'identifier les patrons récurrents, nous avons pensé à utiliser l'analyse formelle de concepts (AFC) [65]. Celle-ci a été intensivement utilisée dans la recherche d'informations et l'extraction de connaissances à partir des données. Elle permet de structurer les données sous une forme de regroupements hiérarchiques (treillis). Comme elle est basée sur la factorisation maximale, elle facilite la compréhension et l'identification des relations entre les données. Elle identifie aussi toutes les répétitions possibles. Plusieurs travaux de recherche en génie logiciel se sont basés sur l'AFC : la remodularisation du code [15, 88, 132], l'identification des objets dans le code procédural [14] et la restructuration des hiérarchies de classes [66, 116, 135, 136]. Elle a été aussi utilisée pour la modularisation du code légataire [15, 132].

Un treillis de concepts permet de représenter les hiérarchies de classes de manière plus explicite. En effet, un treillis construit à partir de l'ensemble des classes de la hiérarchie avec comme propriétés leurs membres associés (aussi bien déclarés qu'hérités) permet de refléter les liens existants de spécialisation entre ces classes sans pour autant les exprimer explicitement. Ces liens sont traduits à travers le type sous-concept-de au sein du treillis. Comme remarqué dans [68], le treillis obtenu d'une hiérarchie de classes de la manière décrite ci-dessus représente une forme détaillée maximale de la hiérarchie. Plus particulièrement, en interprétant les concepts du treillis comme des classes poten-

tielles (dont le type sous-jacent est défini sur la base des membres affectés aux concepts via l'intension), nous définissons un espace de solutions dont les éléments constituent tous les types de classes potentiellement éligibles pour faire partie de la hiérarchie finale. Ainsi, nous jugeons utile de formaliser la tâche de détection des fonctionnalités dans un cadre fondé sur les treillis de concepts.

Une situation très régulière (combinaison exhaustive entre modalités) peut être traitée de manière globale par une décomposition en facteurs. Il s'agira de décomposer le treillis de la hiérarchie de classes initiale en des treillis facteurs dont on pourrait, du moins en théorie, tirer les hiérarchies de classes des fonctionnalités. Dans ce contexte, il est nécessaire d'examiner les divers opérateurs de décomposition de treillis afin de choisir la manière appropriée pour la détection de ce type de fonctionnalités. Ainsi, nous proposons d'explorer ces opérateurs pour identifier les sous-hiérarchies récurrentes. Ceux-ci permettent de décomposer un treillis de concepts en sous-structures pour faciliter l'interprétation des données. Notre objectif est de savoir comment utiliser les algorithmes de décomposition afin d'identifier les différentes fonctionnalités dans du code légataire.

Snelting a été parmi les premiers qui a exploré les algorithmes de décomposition de l'AFC pour la restructuration des hiérarchies de classes et la compréhension du code. Son but était de pouvoir trouver les anomalies dans la conception des hiérarchies de classes [136]. Il a utilisé l'algorithme de décomposition horizontale afin d'avoir des sous-structures du treillis indépendantes [61, 86]. Chaque sous-structure représenterait des parties du code indépendante.

Comme un treillis de concepts est une version étendue de la hiérarchie, toute hiérarchie de classes peut être formalisée et traduite sous forme d'un treillis. Nous allons explorer l'algorithme de décomposition directe/sous-directe car le but est d'identifier les parties récurrentes dans le code qui peuvent être sous forme de structures hiérarchiques.

La suite du chapitre est organisée comme suit : la section 5.3 présente l'ensemble des définitions de base et les notions fondamentales de la théorie de l'analyse formelle des concepts (AFC) [65]. La section 5.4 présente l'algorithme de décomposition sous-directe illustré par quelques exemples facilitant sa compréhension. La dernière section décrit l'utilisation prévue de cet algorithme de décomposition ainsi que deux contributions

négatives à cet égard.

5.3 Principes de l'analyse formelle de concepts

Dans cette partie, nous décrirons les notions de bases de la théorie de l'AFC (Analyse Formelle de Concepts).

5.3.1 Notions de base

5.3.1.1 Ensemble ordonné

Définition 1 (Relation d'ordre partiel)**.** *Soit M un ensemble et $\leq$ une relation binaire sur M. Une relation d'ordre partiel sur M est telle que : pour tout x, y et z de M :*

- *$\leq$ est reflexive : $x \leq x$,*
- *$\leq$ est antisymétrique : si $x \leq y$ et $y \leq x$ alors x=y,*
- *$\leq$ est transitive : si $x \leq y$ et $y \leq z$ alors $x \leq z$.*

Définition 2 (Ensemble ordonné)**.** *(M, $\leq$) est dit ordonné si M est un ensemble et $\leq$ est une relation d'ordre sur M.*

5.3.1.2 Diagramme de Hasse

Un ensemble ordonné (M, $\leq$), peut être représenté par un diagramme composé de nœuds liés par des arrêtes appelé diagramme de Hasse (voir figure 5.3). Les nœuds correspondent aux éléments de M. Pour deux éléments de M, x et y tels que $y < x$, le nœud de y va être représenté au-dessous du nœud de x et les deux nœuds sont liés par une ligne (voir figure 5.3). Un nœud z ne peut être sur la ligne liant x et y si $z \neq x$ et $z \neq y$.

5.3.1.3 Éléments particuliers

Définition 3 (Chaîne et antichaîne)**.** *Deux éléments x, y d'un ensemble partiellement ordonné (M,$\leq$) sont comparables si et seulement si $x \leq y$ ou $y \leq x$, sinon ils sont incom-*

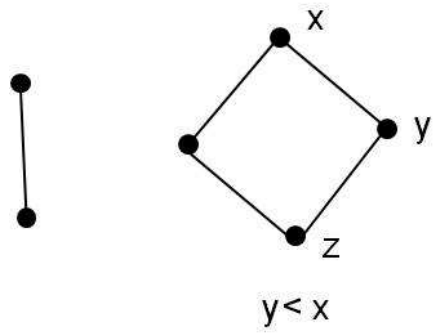


Figure 5.3 – Exemple de Diagrammes de Hasse

parables. Un sous-ensemble de M dans lequel chaque paire d'éléments sont comparables est appelé une chaîne.

Un ensemble A tel que $\forall x, y \in A\ x \leq y \Longrightarrow x = y$*, est appelé* antichaîne.

Exemple : Sur la figure 5.3, l'ensemble $\{x, y, z\}$ représente une chaîne.

Définition 4 (majorant, minorant)**.** *Soient (M,* $\leq$*) un ensemble ordonné et N un sous-ensemble non vide de M. Un élément n est dit un majorant (respectivement un minorant) de N, si pour tout x de N,* $x \leq n$ *(respectivement* $x \geq n$*).*

Définition 5 (infimum, supremum)**.** *Le plus petit élément des majorants, s'il existe, est appelé le supremum noté par* $\vee N$*. De même, le minorant de N est un élément n de M avec* $n \leq p$ *pour tous* $p \in N$*. Le plus grand élément des minorants, s'il existe, est appelé l'infimum noté par* $\wedge N$*.*

Définition 6 (Treillis)**.** *Soit (M,* $\leq$*) un ensemble ordonné. (M,* $\leq$*) est un treillis si pour tout x, y de M :* $(x \wedge y)$ *et* $(x \vee y)$ *existent.*

Exemple : L'ensemble des entiers naturels muni de la relation "divise" forme un treillis, où la borne supérieure est le *PPCM* (supremum) et la borne inférieure est le *PGCD* (infimum). C'est un treillis borné (l'élément minimum est 1 ($\perp$), l'élément maximum est 0 ($\top$)).

Définition 7 (Treillis complet)**.** *Un treillis M est complet si pour toute partie N non vide de M,* $\vee N$ *et* $\wedge N$ *existent.*

Exemple : L'ensemble des entiers naturels est un treillis complet puisque pour toute partie N non vide de l'ensemble, le $\vee$N (*PPMC*) et $\wedge$N (*PGDC*) existent (voir figure 5.4).

divise

A	B	C	D	E	F	G
divise	1	2	3	4	5	6
1	X	X	X	X	X	X
2	0	X	0	X	0	X
3	0	0	X	0	0	X
4	0	0	0	X	0	0
5	0	0	0	0	X	0
6	0	0	0	0	0	X

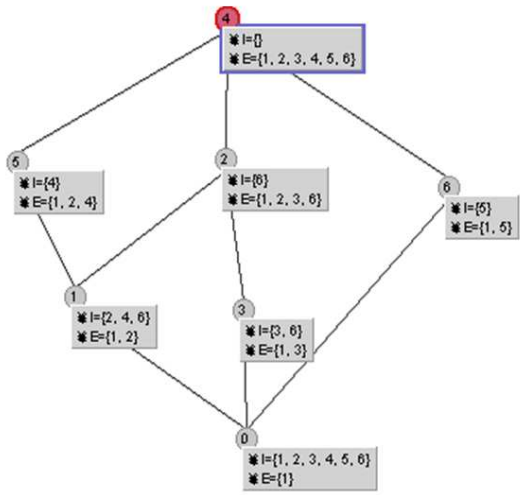


Figure 5.4 – Exemple de treillis complet pour la relation "divise"

Définition 8 (Principe de dualité)**.** *Soit la relation binaire $\leq$ d'un ensemble M vers un ensemble N. La relation inverse notée $\geq$ de la relation $\leq$, est la relation binaire de N vers M telle que :*

$\forall(x,y) \in M \times N, x \leq y \Leftrightarrow y \geq x.$

5.3.2 Notions fondamentales de l'analyse formelle de concepts

L'AFC est une méthode d'analyse de données qui repose sur la notion de concepts formels. Les concepts formels sont munis d'une relation d'ordre partiel. Ils sont ordonnés à travers une structure de treillis complet appelée treillis de concepts. Cette structure est induite par une relation binaire entre deux ensembles, l'ensemble d'objets formels

et l'ensemble d'attributs formels respectivement. Dans ce qui suit, nous allons définir quelques notions fondamentales de cette analyse.

5.3.2.1 Contexte formel

Définition 9 (Contexte formel)**.** *Un contexte formel (O,A,I) est formé de deux ensembles O et A liés par une relation binaire I (ou relation d'incidence). Les éléments de O sont appelés les objets formels et ceux de A les attributs formels. La relation décrit par $(o,a) \in I$ ou oIa, signifie que l'objet o possède l'attribut a.*

Le format de base dans l'AFC [65] est une table binaire appelée *la table de contexte* où les lignes sont associées aux *objets (formels)* (O) et les colonnes à leurs *attributs (formels)* (A). L'intersection cochée entre une ligne et une colonne signifie que l'objet représenté par cette ligne possède l'attribut représenté par cette colonne. Un contexte $\mathscr{K} = (O,A,I)$ est une représentation de la relation d'incidence I.

5.3.2.2 Opérateurs de fermeture

Définition 10 (Opérateurs de fermeture)**.** *Soient deux opérateurs notés par $'$ et $''$ liant les objets et les attributs formels d'un contexte (O, A, I) tels que :*

Soient $X \subseteq O$, $Y \subseteq A$:

- $X' = \{a \in A | \forall o \in X, oIa\}$,
- $Y' = \{o \in O | \forall a \in Y, oIa\}$.

Un sous-ensemble $X \subseteq \mathscr{P}(O)$ est fermé si $X'' = X$(respectivement pour un sous-ensemble d'attributs $Y \subseteq \mathscr{P}(A)$ est fermé si $Y'' = Y$). L'opérateur $''$ représente l'opérateur de fermeture pour $\mathscr{P}(O)$ et $\mathscr{P}(A)$ respectivement.

Reprenons l'exemple de la figure 5.4, $\{3,6\}' = \{1,3\}$ et $(\{3,6\}')' = \{1,3\}' = \{3,6\}$

5.3.2.3 Contexte clarifié

Définition 11 (Contexte clarifié)**.** *Un contexte (O,A,I) est clarifié, si pour chaque paire d'objets g, h $\in$ O avec $g' = h'$, alors g= h (dualement $m' = n'$ implique m=n pour tous m, n $\in$ A).*

Dans un contexte clarifié, il ne pourrait exister deux objets formels différents, tels que les ensembles d'attributs formels associés respectivement sont identiques. Duallement, il ne pourrait exister deux attributs formels différents, tels que les ensembles d'objets formels associés respectivement sont identiques.

5.3.2.4 Contexte réduit

Définition 12 (Contexte réduit)**.** *Un contexte clarifié (O,A,I) est appelé réduit par lignes, si chaque concept objets est $\vee$-irréductible et réduit par colonne, si chaque concept attribut est $\wedge$-irréductible. Un contexte est appelé un contexte réduit s'il est réduit par ligne et par colonne.*

5.3.2.5 Concept formel, infimum et supremum d'un treillis

Définition 13 (Concept Formel)**.** *Un concept formel d'un contexte formel $\mathscr{K} = (O,A,I)$ est une paire (X, Y) où $X' = Y$ et $Y' = X$. L'ensemble X est nommé extension (extents) et Y intension (intents) du concept formel (X,Y). En outre, l'ensemble $\mathscr{C}_{\mathscr{K}}$ de tous les concepts du contexte $\mathscr{K} = (O,A,I)$ est partiellement ordonné par l'inclusion des extents : $(X_1,Y_1) \leq_{\mathscr{K}} (X_2,Y_2)(\Leftrightarrow X_1 \subseteq X_2)$ appelée par la spécialisation entre concepts.*

Définition 14 (Infimum et supremum du treillis)**.** *Pour deux concepts formels (O_1,A_1) et (O_2,A_2) :*

- *L'infimum ou le meet s'exprime par : $(O_1,A_1) \wedge (O_2,A_2) = ((O_1 \cup O_2)'', A_1 \cap A_2)$.*
- *Le supremum ou la jointure est : $(O_1,A_1) \vee (O_2,A_2) = (O_1 \cap O_2, (A_1 \cup A_2)'')$.*

Table 5.I est la table de contexte représentée par la figure 5.5 où *O* est l'ensemble des classes et *A* est l'ensemble des attributs. *I* est la relation entre les deux ensembles.

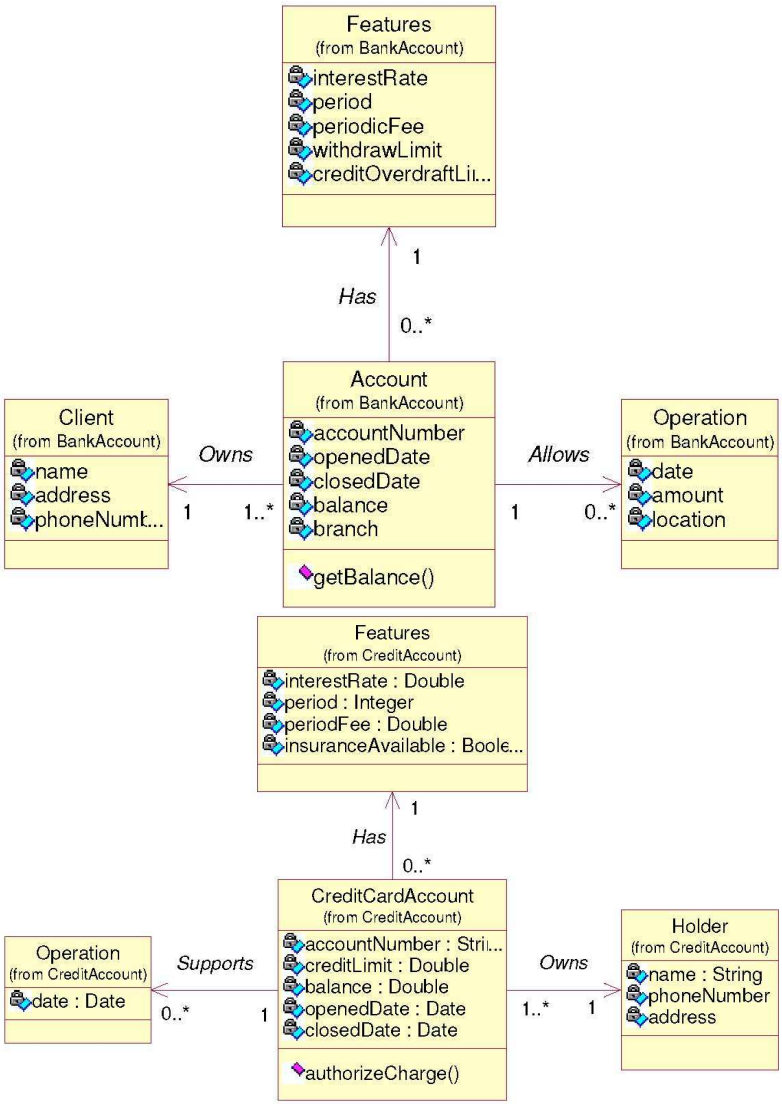


Figure 5.5 – **Haut :**Modèle du domaine d'un compte bancaire. **Bas :** Modèle du domaine d'une carte de crédit.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*	*q*	*r*	*s*
1	X			X		X								X					
2		X											X				X		
3								X			X				X	X			X
4			X						X			X							
5	X			X		X	X							X					
6			X						X									X	
7										X	X				X	X			
8		X											X				X		

Tableau 5.I – La table du contexte $\mathscr{K}$.

Les objets formels sont les classes du modèle UML : *Account (1), Client (2), BA-Features (3), BA-Operation (4), CreditCardAccount (5), CCA-Operation (6), CCA-Features (7), Holder (8)*. Les attributs formels sont les attributs des classes : *accountNumber (a), address (b), amount (c), balance (d), branch (e), closeDate (f), creditLimit (g), creditOverdraftLimit (h), date (i), insurance (j), interestRate (k), location (l), name (m), openedDate(n), period (o), periodicFee (p), phoneNumber (q), units (r), withdrawLimit (s)*.

À partir de la table de contexte illustrée par le tableau 5.I, $\{2,8\}' = \{b,m,q\}$. Dans notre exemple, $(\{2,8\},\{b,m,q\})$ est un concept (voir concept numéro 4 dans la figure 5.6). Un concept peut être défini comme le rectangle maximal dans la matrice de contexte rempli de $'X'$: il est impossible d'augmenter l'intent sans réduire l'extent et *vice versa*.

Théorème 1 (Théorème de base sur les treillis de concepts)**.** *$\mathscr{L} = \langle \mathscr{C}_{\mathscr{K}}, \leq_{\mathscr{K}} \rangle$ est un treillis complet, appelé le treillis de Galois ou de concepts, où les bornes inférieures et les bornes supérieures sont respectivement :*

- $\bigvee_{i=1}^{k}(X_i,Y_i) = ((\bigcup_{i=1}^{k} X_i)'', \bigcap_{i=1}^{k} Y_i)$
- $\bigwedge_{i=1}^{k}(X_i,Y_i) = (\bigcap_{i=1}^{k} X_i, (\bigcup_{i=1}^{k} Y_i)'')$.

Le treillis $\mathscr{L}$ peut être présenté par un diagramme de Hasse où les nœuds sont les concepts composés des extents (ensembles des objets formels) et des intents (ensemble

d'attributs formels). L'arrête liant les concepts est la ligne de spécialisation entre ces concepts. Le treillis du contexte associé à la table de contexte du tableau 5.I est représenté par la figure 5.6. Une représentation simplifiée est souvent utilisée pour que chaque objet et chaque attribut apparaît une seule fois sur le diagramme.

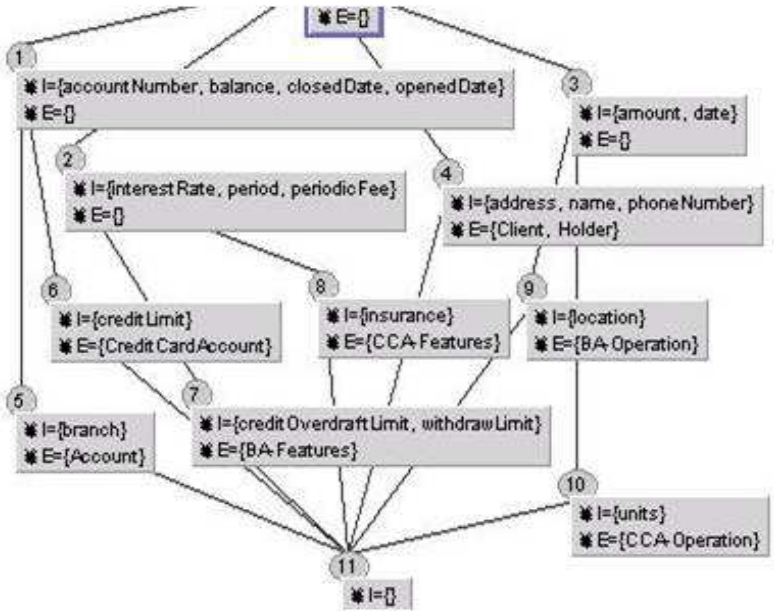


Figure 5.6 – Le treillis du contexte lié au compte bancaire.

Propriété 1 (Treillis de concepts)**.** *L'ensemble de tous les concepts formels d'un contexte* $\mathscr{K} = (O, A, I)$ *est un treillis complet.* $\mathscr{L}$ *est le treillis de concepts du contexte* $\mathscr{K}$*. Pour tous les sous-ensembles de concepts X de* $\mathscr{L}$*, le supremum* $\vee X$ *et l'infimum* $\wedge X$ *existent.*

Pour un concept formel C, ext(C) signifie l'extension de C ou les objets du concept C et int(C) signifie l'intension de C ou les attributs de C.

Dans le pire cas, le treillis de Galois peut croître exponentiellement avec le nombre d'objets et de propriétés. Cependant, lorsqu'il y a une borne supérieure, disons k, la taille du treillis est bornée linéairement par rapport au nombre d'objets n tel que : $\mathscr{L} \leq 2^k n$ [67].

Définition 15 (Les concepts top et bottom d'un treillis)**.** *Le concept formel maximal d'un*

treillis $\mathscr{L}$ est appelé le top *du treillis, noté $\vee\mathscr{L}$ ou $\top$. Le concept formel minimal d'un treillis est appelé le* bottom *du treillis, noté $\wedge\mathscr{L}$ ou $\bot$.*

Intuitivement, en descendant du top au bottom, les concepts possèdent de moins en moins d'objets et partagent de plus en plus d'attributs.

5.3.2.6 Quelques notions utilisées pour la décomposition de treillis

Définition 16 (Fonctions auxiliaires)**.** *Le plus grand concept où $a \in int(C)$ est noté $\mu(a)$ avec a un élément de l'ensemble des attributs des concepts d'un treillis. $\mu(a) = \vee_{a\in int(C)} C = (a', a'')$. Dualement, $\gamma(o)$ pour un élément o de l'ensemble des objets des concepts d'un treillis, est le plus petit concept où $o \in ext(C)$. $\gamma(o) = \wedge_{o\in ext(C)} C = (o'', o')$.*

Dans le treillis de la figure 5.4, pour l'attribut 6, $\mu(6) = (6', 6'') = \{1,2,3,6\}$ et pour l'objet 3, $\gamma(3) = (3'', 3') = (\{1,3\},\{3,6\})$. $\mu(6)$ est représenté par le concept 2 et $\gamma(3)$ est représenté par le concept 3.

Définition 17 (Congruence de treillis)**.** *Une relation de congruence Θ dans un treillis complet $\mathscr{L}$ est une équivalence qui respecte la propriété de substitution :*

- *$x_1 \Theta x_2$ et $y_1 \Theta y_2 \Longrightarrow (x_1 \vee y_1)\Theta(x_2 \vee y_2)$ et $(x_1 \wedge y_1)\Theta(x_2 \wedge y_2)$.*

Proposition 1. *Une relation de congruence d'un treillis $\mathscr{L}$ est une relation d'équivalence θ sur $\mathscr{L}$, pour t variante dans T (un ensemble d'indexes) :*

$$x_t \theta y_t \Longrightarrow (\bigvee_{t\in T} x_t)\theta(\bigvee_{t\in T} y_t) et (\bigwedge_{t\in T} x_t)\theta(\bigwedge_{t\in T} y_t).$$

Proposition 2. *chaque classe de congruence $[]_\Theta$ représente un intervalle au sein du treillis,*

La classe d'équivalence x est : $[x]_\theta = \{y \in \mathscr{L} / x\theta y\}$.

$\mathscr{L}|_\theta = [x]_\theta / x \in \mathscr{L}$ est appelé **treillis facteur**. Il possède la relation d'ordre suivante :

$$[x]_\theta \leq [y]_\theta \Longleftrightarrow x\theta(x \wedge y) \Longleftrightarrow (x \vee y)\theta y$$

Une congruence est induite par un sous-contexte sous certaines conditions.

Définition 18 (Projection canonique)**.** *Tout sous-contexte* $(H,N,I\cap H\times N)$ *d'un contexte* (O,A,I) *définit une projection canonique* $\Pi_{H,N}$ *sur l'ensemble des concepts* $\mathscr{B}(O,A,I)$ *:*

- $\Pi_{H,N} : \wp(O)\times\wp(A)\to\wp(H)\times\wp(N)$,
- $\Pi_{H,N}(X,Y) = (X\cap H, Y\cap N)$.

Définition 19 (Sous-contexte compatible)**.** *Un sous-contexte* $(H,N,I\cap H\times N)$ *d'un contexte* (O,A,I) *réduit est dit compatible si pour tout concept* (X,Y) *de* (O,A,I), $(X\cap H, Y\cap N)$ *est un concept de* $(H,N,I\cap H\times N)$.

Autrement dit, un sous-contexte $(H,N,I\cap H\times N)$ *est compatible si pour chaque concept* $(A,B)\in\mathscr{L}$, $(A\cap H, B\cap N)$ *est un concept du sous-contexte* $(H,N,I\cap H\times N)$.

$(H,N,I\cap H\times N)$ *de* (O,A,I) *est compatible ssi la projection :*
$\Pi_{H,N} : \mathscr{B}(O,A,I)\to\mathscr{B}(H,N,I\cap H\times N)$.
$\Pi_{H,N}(A,B) = (A\cap H, B\cap N)$, *pour tout* $(A,B)\in\mathscr{L}$, *est un homomorphisme surjectif.*

Proposition 3. *Si un sous-contexte* $(H,N,I\cap H\times N)$ *d'un contexte* (O,A,I) *réduit est compatible, alors la projection canonique* $\Pi_{H,N}$ *est un homomorphisme [65], c'est-à-dire :*

- $\Pi_{H,N}((X_1,Y_1)\vee_{O,A}(X_2,Y_2)) = \Pi_{H,N}(X_1,Y_1)\vee_{H,N}\Pi_{H,N}(X_2,Y_2)$,
- $\Pi_{H,N}((X_1,Y_1)\wedge_{O,A}(X_2,Y_2)) = \Pi_{H,N}(X_1,Y_1)\wedge_{H,N}\Pi_{H,N}(X_2,Y_2)$.

Proposition 4. *Tout homomorphisme d'algèbre induit une congruence [65],*

Définition 20 (Relations flèches)**.** *Les relations flèches particularisent le complément de l'incidence I et traduisent des rapports de maximalité :*

- $o\swarrow a$ *ssi* $o \not I a$ *et* $\forall h\in O$, $o'\subset h'$ *implique* hIa,
- $o\nearrow a$ *ssi* $o \not I a$ *et* $\forall n\in A$, $a'\subset n'$ *implique* oIn,
- $o\swarrow\!\!\!\nearrow a$ *ssi* $o\swarrow a$ *et* $o\nearrow a$.

$o \swarrow a$ est équivalent à dire que o' est maximal parmi tous les intents ne contenant pas a. En d'autres termes, $o \swarrow a$ ssi o ne possède pas l'attribut a, mais a est parmi les attributs de chaque sous-concept (o'', o'). Le même raisonnement s'applique pour $o \nearrow a$ mais d'une manière duale [65].

Définition 21 (Contexte fermé par flèches)**.** *Un sous-contexte* $(H, N, I \cap H \times N)$ *de* (O, A, I) *est dit fermé par flèches si :*

- $h \swarrow a$ *et* $a \in N$*, implique* $h \in H$*,*
- $o \nearrow n$ *et* $o \in H$ *implique* $n \in N$*.*

Proposition 5. *Si* (O, A, I) *est réduit, alors tout sous-contexte* $(H, N, I \cap H \times N)$ *fermé par flèches est aussi compatible.*

Pour un contexte $\mathscr{K} = (O, A, I)$, $H \subseteq O$ et $N \subseteq A$. $(H, N, I \cap H \times N)$ est le sous-contexte de $\mathscr{K}$. L'ensemble de concepts formels d'un contexte $\mathscr{K}$ est $\mathscr{L} = \mathscr{B}(O, A, I)$.

Un sous-contexte compatible $(H, N, I \cap H \times N)$ de $\mathscr{K}$ est isomorphe à un treillis facteur de $\mathscr{L}$:

$$\mathscr{B}(H, N, I \cap H \times N) \cong \mathscr{L}|_{\theta_{\mathscr{H},\mathscr{N}}}$$
$$(A, B)\theta_{H,N}(A', B') \iff A \cap H = A' \cap H$$
$$\iff B \cap N = B' \cap N$$

Soit un exemple illustré par la table de contexte du tableau 5.II. Nous appliquons les relations flèches sur celle-ci afin de pouvoir extraire des sous-contextes compatibles. Le tableau 5.III reflète la génération des relations flèches sur la table de contexte initiale présentée dans le tableau 5.II. Le graphe dirigé des relations flèches est illustré par la figure 5.7. À partir du graphe dirigé, nous obtenons les sous-contextes compatibles. Les sous-contextes fermés par flèches déduits de la table 5.III après l'application des relations flèches sont : $(\{1, 2, 3\}, \{a, b, c, d\})$ et $(\{4, 5\}, \{e, f\})$ (voir figure 5.7).

I	*a*	*b*	*c*	*d*	*e*	*f*
1	X	X			X	X
2		X	X			X
3			X	X		X
4	X	X	X	X		X
5			X	X		

Tableau 5.II – Exemple de table de contexte

I	*a*	*b*	*c*	*d*	*e*	*f*
1	X	X	↗↙	↙	X	X
2	↗↙	X	X	↗↙		X
3	↙	↗↙	X	X		X
4	X	X	X	X	↗↙	X
5			X	X		↗↙

Tableau 5.III – Relations flèches appliquées sur la table 5.II

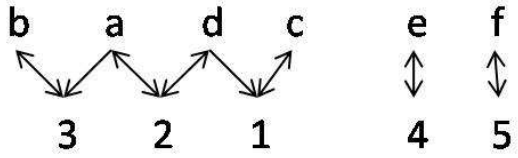


Figure 5.7 – Graphe dirigé correspondant aux relations flèches de la table 5.II

5.3.2.7 Produit direct

Définition 22 (Produit Tensoriel [65]). *Le **produit tensoriel** de deux treillis $\mathscr{L}_1$ et $\mathscr{L}_2$, est défini par :*
$\mathscr{L}_1 \otimes \mathscr{L}_2 = \mathscr{B}(\mathscr{L}_1 \times \mathscr{L}_2, \mathscr{L}_1 \times \mathscr{L}_2, \nabla)$
$(x_1,x_2)\nabla(y_1,y_2) \Leftrightarrow x_1 \leq y_1$ *ou* $x_2 \leq y_2$ *pour* $(x_1,x_2),(y_1,y_2) \in \mathscr{L}_1 \times \mathscr{L}_2$.

Définition 23 (Produit Direct [143]). *Le **produit direct** de deux contextes $\mathscr{K}_1 = (O_1,A_1,I_1)$ et $\mathscr{K}_2 = (O_2,A_2,I_2)$ est défini par :*
$\mathscr{K}_1 \times \mathscr{K}_2 = (O_1 \times O_2, A_1 \times A_2, \nabla)$
$(o_1,o_2)\nabla(a_1,a_2) \Leftrightarrow o_1 I_1 a_1$ *ou* $o_2 I_2 a_2$ *pour* $(o_1,o_2) \in O_1 \times O_2$ *et* $(a_1,a_2) \in A_1 \times A_2$.

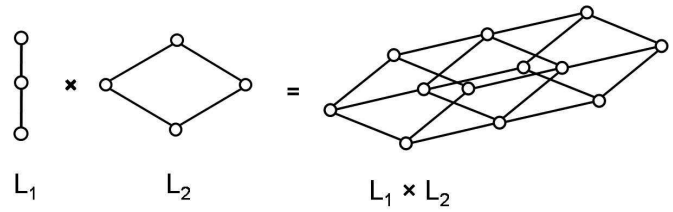


Figure 5.8 – Produit direct de deux treillis

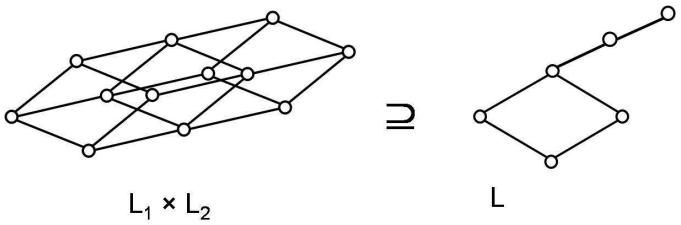


Figure 5.9 – Décomposition sous-direct

Le produit direct est le produit de deux contextes. L, de la figure 5.9, est un sous-treillis de $L_1 \times L_2$. L_1, L_2 peuvent être obtenus par la décomposition sous-directe de L.

Proposition 6. *Le treillis de concepts d'un produit direct de contextes est isomorphe au produit tensoriel de treillis de concepts [65].*

5.4 Algorithme de décomposition sous-directe de treillis

Le produit direct de facteurs est une superposition parfaite des classes des différents facteurs, pour avoir une combinaison de toutes les classes deux à deux. Ceci amène les fonctionnalités a être superposées. Or, dans les applications réelles, nous ne trouvons pas cette superposition parfaite. Par suite, la décomposition directe n'est pas plausible. Il est rare qu'une hiérarchie conçue manuellement regroupe toutes les combinaisons de classes, deux à deux, émanant des hiérarchies d'origine.

Comme en pratique la combinaison des classes n'est pas toujours parfaite et complète, pour alléger les contraintes de compositions, nous présentons dans la partie qui suit l'algorithme de décomposition sous-directe de treillis afin de fractionner le treillis sous forme de treillis facteurs.

La décomposition sous-directe est prometteuse pour la restructuration (pour les détails voir [59]). Elle est basée sur les relations flèches. Ces relations sont introduites dans la table du contexte sur les attributs et les objets formels qui ne sont pas en relation par la relation binaire. Le graphe constitué par $(O \cup A, \nearrow \cup \swarrow)$ est nommé graphe dirigé (directed graph). Les sous-contextes compatibles sont formés à partir des composantes fermées par les relations flèches (arrow-closed) du graphe dirigé [65]. Les sous-contextes compatibles sont utilisés comme projections sur le treillis initial afin d'obtenir les treillis facteurs. L'ensemble des concepts d'un tel treillis facteur représente la classe d'équivalence qui permet d'identifier un treillis facteur.

Exemple d'application de la décomposition sous-directe

Nous allons appliquer la décomposition sous-directe sur l'exemple d'application précédent. Soit la table de contexte du tableau 5.II. Le treillis correspondant est présenté par la figure 5.10 [65].

Comme cité auparavant, à partir du graphe, nous avons trouvé deux sous-contextes compatibles $(\{1,2,3\},\{a,b,c,d\})$ et $(\{4,5\},\{e,f\})$ (voir figure 5.7). En appliquant les congruences associées à chacun de ces sous-contextes respectivement, nous obtenons les décompositions présentées par la figure 5.11. Chaque décomposition reflète un treillis facteur. Nous supposons qu'un sous-treillis, qui est le treillis facteur par la décomposition

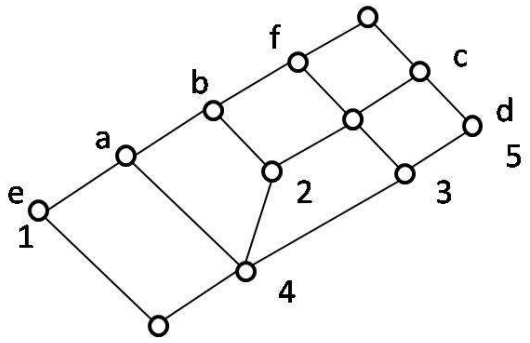


Figure 5.10 – Treillis correspondant à la table 5.II

sous-directe, représentera un ensemble de concepts cohésifs. Par suite, la décomposition sous-directe sera étudiée et explorée.

5.5 Conclusion

L'Analyse Formelle de Concepts (AFC) a été intensivement utilisée dans la recherche d'informations et l'extraction de données [66, 116, 135, 136] [15, 132]. Dans ce chapitre, notre objectif était d'étudier les algorithmes de décomposition pour pouvoir les utiliser pour l'identification des facteurs exprimant différentes fonctionnalités existantes. Malheureusement dans le chapitre qui suit, nous allons démontré à travers deux preuves détaillées que dans le cas du produit parfait, un sous-treillis ne peut être que le treillis global initial.

Notre deuxième approche consiste à envisager une autre manière de définir la relation binaire pour générer les treillis de Galois pour identifier les différentes fonctionnalités. Les détails de cette approche sont présentés dans le chapitre qui suit. Nous allons décrire et formaliser notre nouvelle relation binaire afin d'identifier facilement les différentes fonctionnalités récurrentes dans un code légataire orienté objets même dans le cas parfait de multiplication d'états.

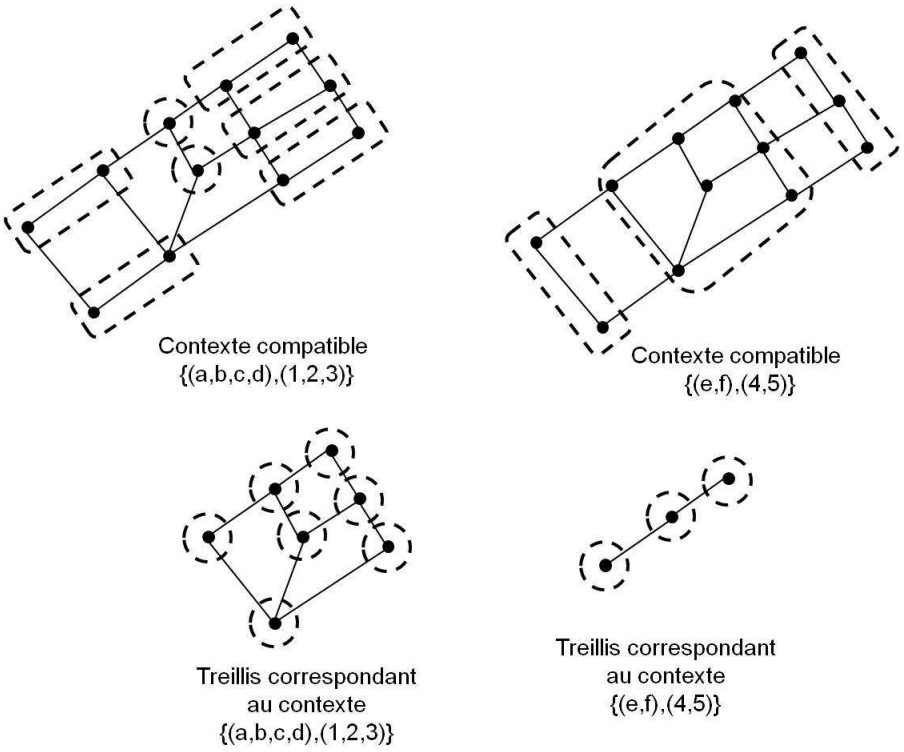


Figure 5.11 – Décomposition sous-directe de la deuxième congruence

CHAPITRE 6

ALGORITHMES D'IDENTIFICATION DES FONCTIONNALITÉS PAR MULTIPLICATION D'ÉTATS

Comme déjà décrit dans les chapitres précédents, nous définissons une fonctionnalité par une occurrence multiple de données et/ou des méthodes dans différents endroits de la hiérarchie indépendamment de sa structure. Dans l'exemple de la figure 5.1, la même fonctionnalité se trouve dans deux sous-hiérarchies. Elle est représentée par l'ensemble {*schedule, capabilities, assemblyLine, licenseClass*}. L'attribut *capabilities* est localisé dans un niveau différent par rapport à la sous-hiérarchie située à droite. Comme mentionné dans le chapitre précédent, la structure de la hiérarchie ne peut être considérée ici comme un critère d'identification puisque l'ensemble d'attributs est le même, mais la répartition est différente. En effet, elle ne peut couvrir toutes les occurrences des candidats puisqu'il suffit que le même ensemble d'attributs se présente dans un autre endroit de la hiérarchie avec un attribut situé dans un emplacement différent (voir l'explication et l'illustration à travers l'exemple de la figure 5.2 dans le chapitre 5).

Puisque l'analyse formelle de concepts (AFC) est reconnue pour sa capacité de factorisation maximale des données, nous avons opté pour l'utilisation des treillis de concepts pour identifier ces occurrences multiples. Dans ce qui suit, nous allons motivé notre décision d'utiliser l'AFC afin de pouvoir identifier les différentes occurrences multiples du cas de multiplication d'états dans le code. Nous allons supposé que ces occurrences représentent des facteurs récurrents, et ceci nous amène à explorer les algorithmes de décomposition directe/sous-directe.

La figure 6.1 montre le déroulement de notre démarche. Nous allons exploré l'algorithme de décomposition directe. Un tel type d'algorithme est plus strict et exige une structure très régulière pour pouvoir décomposer le treillis. Hélas, ce cas s'est avéré quasiment inexistant dans la pratique. Nous allons étudié par la suite la possibilité d'appliquer la décomposition sous-directe. L'idée était de pouvoir alléger les contraintes afin de nous aider à identifier plus facilement ces occurrences. Même avec cette hypothèse,

nous n'avons pu atteindre notre objectif.

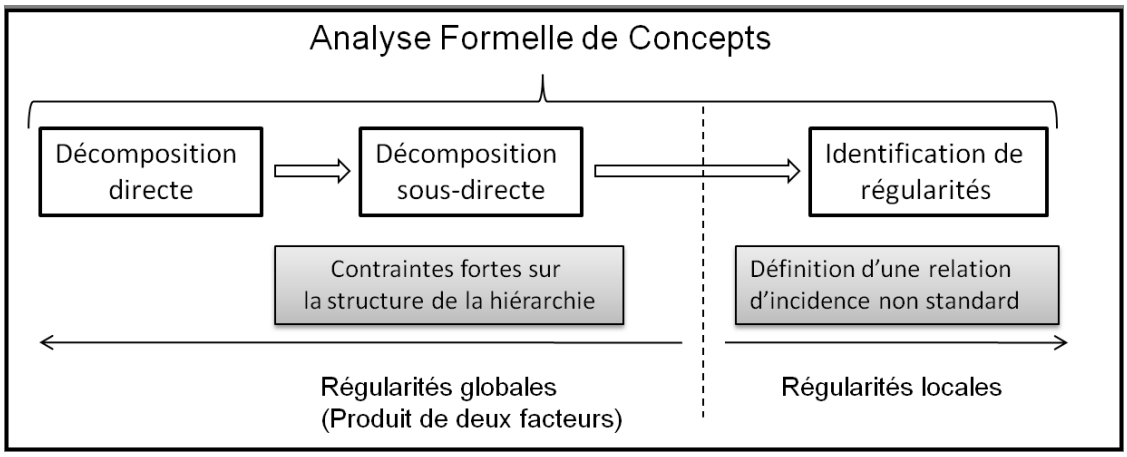


Figure 6.1 – La démarche de notre deuxième approche

6.1 Identification des fonctionnalités à partir de treillis facteurs

L'AFC a contribué, par plusieurs opérateurs, à la composition et la décomposition des treillis. Les algorithmes de décomposition sont utiles pour résoudre le problème de la rétro ingénierie du code et sa compréhension puisqu'ils nous permettent la décomposition d'un treillis global d'un code source, associé à une hiérarchie initiale, en treillis facteurs associés aux hiérarchies récurrentes. Ces treillis facteurs peuvent correspondre à des aspects fonctionnels ou fonctionnalités.

6.1.1 Fonctionnalités composés par multiplication d'états

La technique de multiplication d'états correspond au cas où différentes fonctionnalités récurrentes se trouvent dans une même hiérarchie de classes. Elle illustre la situation qui combine plusieurs fonctionnalités (voir dans le chapitre 3, la section 3.4). Reprenons l'exemple de la figure 3.7. Il existe différents types de carrosseries de voitures, type berline et type familiale. Elles peuvent aussi avoir différents catégories de moteur.

Nous présentons la notion de multiplication d'états comme des ensembles de membres situés dans plusieurs endroits de la hiérarchie. Cette notion se manifeste par le nombre

d'occurrences des membres (attributs formels ou méthodes dans notre cas) de fonctionnalités à différentes places sans être réellement factorisées. Dans la figure 3.7.d, l'attribut "compressionRatio" est défini dans deux nœuds feuilles différents de la hiérarchie. Pour une bonne factorisation de l'attribut, il est suggéré de :

1. le définir à une seule place ;
2. le rendre accessible par "TurboI4Sedan" et "TurboI4StationWagon", ce qui nous amène à le définir dans la classe "Car".

Cependant, il n'est pas concevable de mettre "compressionRatio" pour les voitures de moteur V6 non turbo. C'est habituellement un dilemme pour les concepteurs non expérimentés ou par manque d'attention qui implémentent avec des langages qui ne supportent pas l'héritage multiple.

Graphiquement, un cas de multiplication d'états ressemblerait à un certain produit de hiérarchies (hiérarchies de fonctionnalités) sous-jacentes. Dans notre exemple, hiérarchies de type carrosserie et moteur. Supposons que nous présentons les hiérarchies de fonctionnalités avec un ordre partiel $F_1 = (V_1, R_1)$ et $F_2 = (V_2, R_2)$, où V_1 et V_2 représentent l'ensemble de valeurs de F_1 et F_2, respectivement, et R_1 et R_2 les relations d'ordres partiels pour ces valeurs. Le produit de F_1 et F_2 est défini comme suit :

$$\begin{aligned}(\forall < x,y >, < x^{'}, y^{'} >: x, x^{'} \in V_1 et y, y^{'} \in V_2)\\ << x,y >, < x^{'}, y^{'} >> \in R_{1\otimes 2} \Leftrightarrow\\ ((< x, x^{'} > \in R_1) \vee (< y, y^{'} > \in R_2))\end{aligned}$$

Le produit de hiérarchies des fonctionnalités de la figure 3.7.a et la figure 3.7.b est illustré par la figure 6.2.

Nous utilisons *"Cha"* pour Chassis, *"eng"* pour moteur, *"sed"* pour berline, *"SW"* pour familiale, *"I4"* et *"V6"* pour les types de moteur à 4 cylindres et 6 cylindres respectivement.

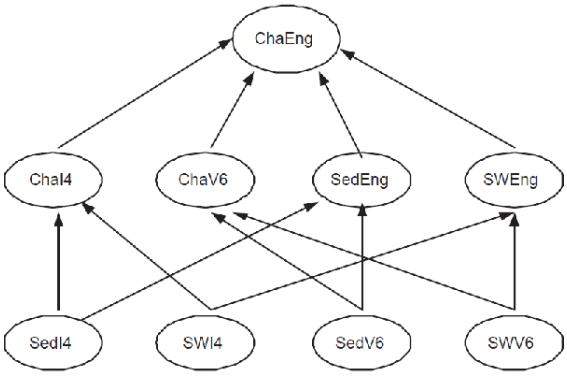


Figure 6.2 – Produit direct des deux hiérarchies

Les fonctionnalités (deux dans notre cas) se retrouvent dans les sous-classes. Dans les applications légataires, malheureusement, il est rare de trouver des hiérarchies de classes qui sont sous forme de produit tensoriel. Par exemple, les hiérarchies des figures 3.7.c et 3.7.d représentent des sous-ensembles du produit de la figure 6.2.

Formulons l'hypothèse que nous pouvons décomposer un treillis global, correspondant à une hiérarchie globale qui représente le produit de deux hiérarchies initiales, afin d'obtenir des treillis facteurs. Chacun de ces treillis facteurs correspond à l'une des hiérarchies initiales respectivement (voir figure 6.3).

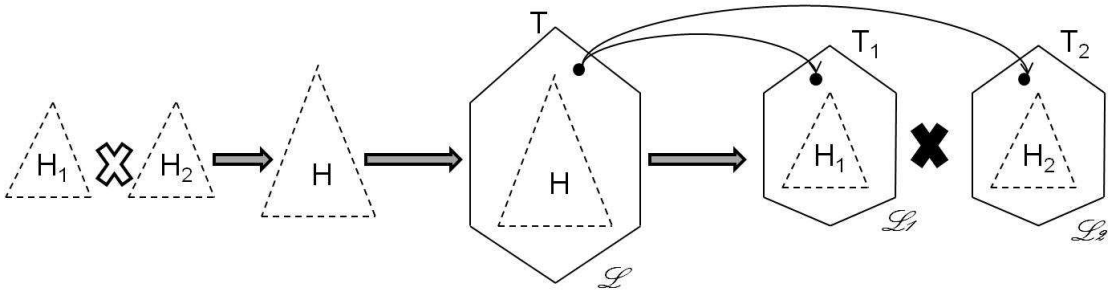


Figure 6.3 – Identification des facteurs treillis qui correspondent à des facteurs de hiérarchies

A travers les opérateurs de l'AFC, le produit direct semble le plus prometteur pour

le cas de multiplication d'états. Le produit *direct* comprend toutes les combinaisons de classes. Par contre, dans l'exemple de la figure 3.7, le treillis obtenu de la hiérarchie produit ne peut être un produit direct des treillis facteurs tiré des hiérarchies facteurs. Il suffit qu'une combinaison de treillis soit manquante dans la hiérarchie globale pour ne pas refléter le produit direct. Par exemple, si la classe *Turbo4Sedan* manquait alors on ne pourrait pas la reproduire dans le treillis de la hiérarchie globale que nous essayons de décomposer.

Aussi en pratique, les instances de multiplication d'états ne correspondent pas aux factorisations parfaites, pour deux raisons :

1. La rareté du domaine : dans l'exemple de la figure 6.2, toutes les combinaisons du type de carrosserie et moteur sont possibles. Dans la vie réelle, plusieurs combinaisons de fonctionnalités ne sont pas possibles techniquement. Parfois, telles combinaisons sont jugées non intéressantes et non faisable. Dans notre exemple, il n'est pas courant qu'un client opte pour une familiale avec un moteur de type turbo.

2. Généralement, les factorisations parfaites n'existent pas dans les combinaisons manuelles de hiérarchies. Il est très rare de trouver toutes les combinaisons des classes. Cette manière de faire est reliée au codage du développeur.

Ainsi, le produit direct est plus exigeant. Chaque concept du treillis global doit correspondre à toute combinaison des deux hiérarchies, soit deux concepts de chacun des treillis facteurs. Chaque classe feuille d'une hiérarchie soit combinée avec toutes les classes de l'autre y compris les racines. Or, ce n'est pas le cas de notre exemple (les classes feuilles de la figure 3.7.a avec les racines de la figure 3.7.b ou inversement mais pas les deux). Ainsi, notre treillis global ne pourra pas être décomposable par le produit direct pour obtenir les treillis facteurs compatibles avec les hiérarchies initiales des fonctionnalités puisque, comme mentionné ci-dessus, le produit direct exige que toutes les combinaisons soient présentes. Par suite, le produit direct est écarté.

Il faut maintenant trouver une décomposition qui n'exige pas toutes les combinaisons. Dans la vie réelle, il existe certaines combinaisons qui pourront être manquante

(une classe feuille avec une autre classe qui n'est pas une classe feuille). La question qui se pose est si nous pouvons modéliser notre exemple avec un autre produit. Nous allons explorer le produit sous-direct puisqu'il n'exige pas la présence de toutes les combinaisons (certaines combinaisons peuvent être exclues). Alors, peut-on espérer de modéliser notre exemple par un produit sous-direct ?

A l'inverse du produit direct de treillis, le sous-direct ne comprend pas toutes les combinaisons de classes de la fonctionnalité. En fait, il représente un sous-treillis du produit direct de deux treillis facteurs où les deux structures de ces treillis facteurs restent préservées. Dans la section qui suit, nous allons montrer à travers deux contributions la non possibilité d'obtenir des treillis facteurs à partir d'un treillis global par l'algorithme de décomposition sous-directe.

6.1.2 Contribution et Preuves

Nous définissons une fonctionnalité par un ensemble d'attributs/méthodes. Un ensemble d'attributs représente une fonctionnalité s'il est récurrent dans la hiérarchie. C'est pourquoi, nos preuves seront limitées sur les attributs formels (les attributs formels dans notre relation binaire, sont les signatures des méthodes).

Considérons la relation binaire I défini entre un objet o et un attribut a comme suit : $(o,a) \in I$ ssi o possède a si a est un attribut propre à o ou c'est un attribut hérité de ses super classes.

6.1.2.1 Preuve 1

Dans le cas de multiplication d'états, le plus grand défi est de trouver une manière qui nous permet de décomposer un treillis en sous-treillis facteurs significatifs. La décomposition sous-directe est utilisée pour identifier ces sous-treillis. Nous supposons que de tels sous-treillis représentent des treillis facteurs où chacun peut illustrer une ou plusieurs fonctionnalités dans le code.

Supposons que toutes les combinaisons des classes feuilles sont présentes. Montrons qu'un sous-treillis du produit qui contient toutes les combinaisons des classes feuilles,

couvre automatiquement la hiérarchie globale.

Soit $\mathscr{L}$ un treillis global. Supposons que la hiérarchie globale est le produit de deux hiérarchies initiales. Le but est de pouvoir identifier des facteurs treillis qui correspondent aux facteurs de hiérarchies initiales respectivement. Soient les hiérarchies de classes suivantes $\mathscr{H}$, $\mathscr{H}_1$ et $\mathscr{H}_2$, où $\mathscr{H}$ est le produit de $\mathscr{H}_1$ et $\mathscr{H}_2$. Générons les treillis associés à chacune de ces hiérarchies. soient $\mathscr{L}$ le treillis associé à $\mathscr{H}$, $\mathscr{L}_1$ le treillis associé à $\mathscr{H}_1$ et $\mathscr{L}_2$ le treillis associé à $\mathscr{H}_2$.

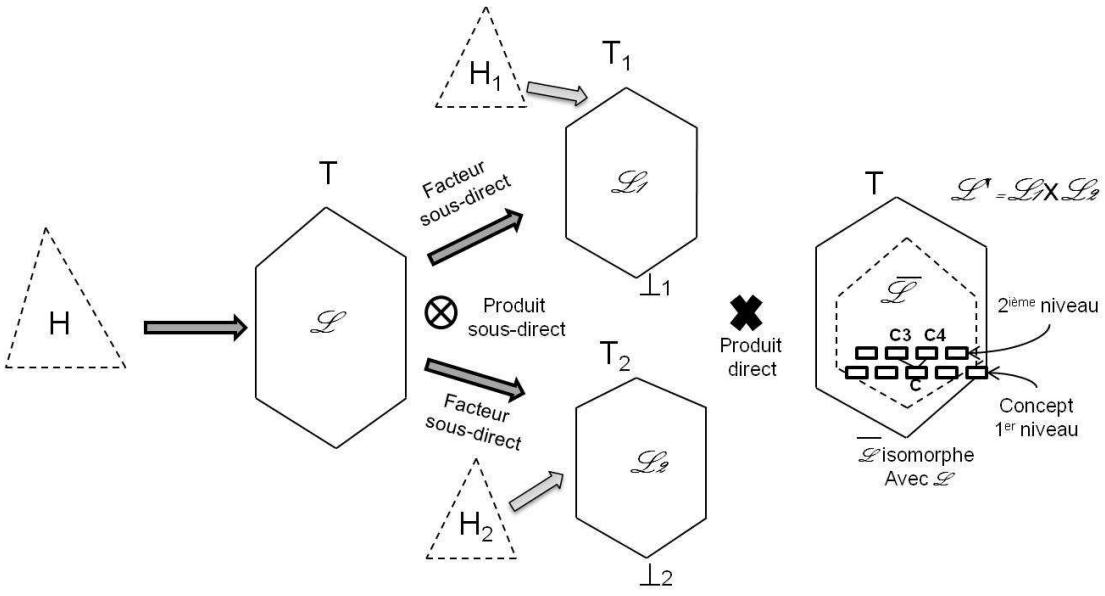


Figure 6.4 – Schéma illustrant l'identification d'un sous-treillis isomorphe au treillis initial

Montrons par raisonnement inductif que dans $\mathscr{L}'$, qui est le produit de $\mathscr{L}_1$ et $\mathscr{L}_2$, nous pouvons trouver $\overline{\mathscr{L}}$ où $\overline{\mathscr{L}}$ est un treillis isomorphe à $\mathscr{L}$. Une propriété fondamentale de la théorie du treillis, le supremum de chaque paire de concepts du sous-treillis doit appartenir au sous-treillis. Par suite, le supremum de chaque paire d'éléments de $\overline{\mathscr{L}}$ doit être dans $\overline{\mathscr{L}}$:

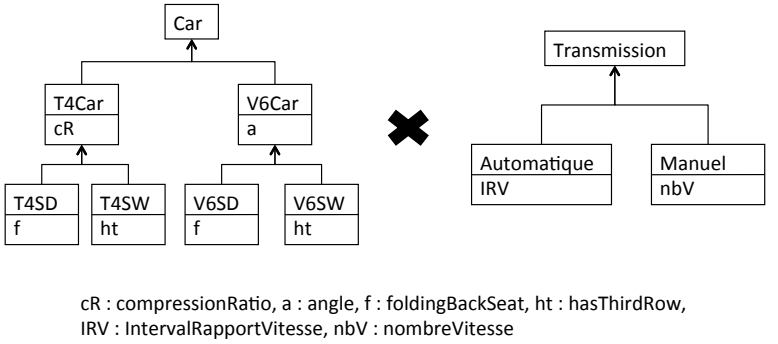
$\forall c \in \mathscr{L}'$ où c est un concept du premier niveau (juste au-dessus du bottom) de $\mathscr{L}', c = c_1 \times \perp_2$ ou $c = \perp_1 \times c_2$, $c_1 \in \mathscr{L}_1$ et $c_2 \in \mathscr{L}_2$.

Soit le cas où $c = c_1 \times \perp_2$.

c est un concept de premier niveau de $\mathscr{L}'$, alors $\exists c_3$ et $c_4 \in \overline{\mathscr{L}}$ tel que :

$c_3 = c_1 \times c_5, c_4 = c_1 \times c_6$, avec $c_5, c_6 \in \mathscr{L}_2$.
Alors, $c = c_3 \vee_{\mathscr{L}'} c_4 \Longrightarrow c \in \overline{\mathscr{L}}$.

Par suite, tout concept de treillis produit direct se retrouve dans le sous-treillis et inversement. Vérifions notre preuve sur l'exemple de la figure 3.7. Nous considérons la relation binaire *I* où chaque classe possède ses propres attributs ainsi que les attributs de tous ses supers classes. Ainsi, en générant le treillis de la hiérarchie globale, les concepts feuilles du treillis combinent l'ensemble de tous les attributs des classes (voir l'exemple présenté par les figure 6.5 et figure 6.6).



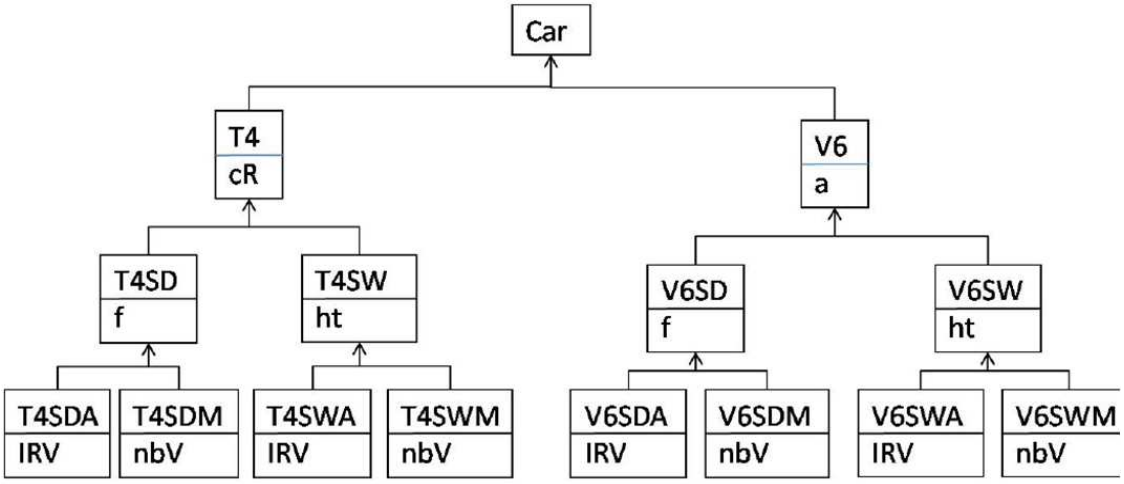


Figure 6.5 – **Haut :** Exemple de produit de deux hiérarchies de classes (**Bas :**) Diagramme de classes résultant du produit

Pour chaque paire de concepts de la base, le supremum est généré et ajouté au sous-treillis. En générant les supremums de tous les concepts, nous atteignons tous les nœuds

du treillis global à cause que, comme cité auparavant, les concepts feuilles combinent tous les attributs de la hiérarchie globale. Au fur et à mesure que la base du sous-treillis $\overline{\mathscr{L}}$ s'élargit (combinaison des nœuds feuilles), l'élément top $\overline{\top}$ du sous-treillis $\overline{\mathscr{L}}$ atteint l'élément top $\top'$ du treillis global $\mathscr{L}'$. Par conséquent, dans le cas parfait de multiplication d'états, le sous-treillis n'est simplement que le treillis initial du produit direct des hiérarchies. Il n'est pas possible d'identifier un sous-treillis facteur. Ce qui nous amène à dire que dans ces conditions et avec telle relation binaire, le treillis est non décomposable.

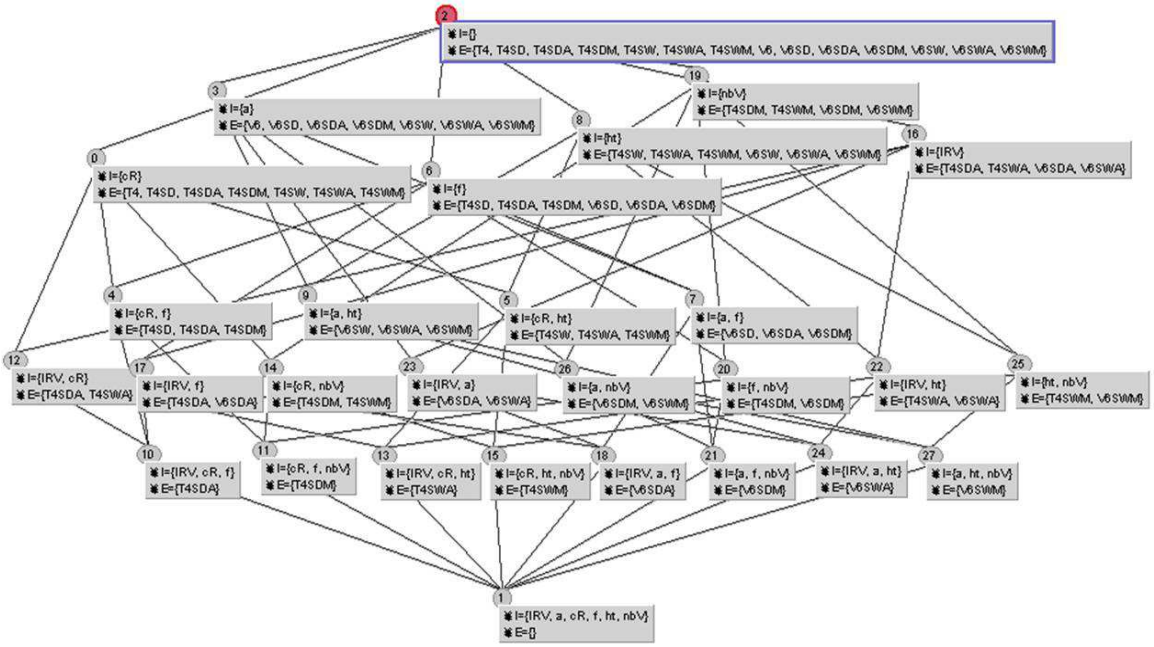


Figure 6.6 – Treillis correspondant au diagramme de classe de figure 6.5

6.1.2.2 Preuve 2

Dans cette section, nous présentons une autre façon de prouver qu'un treillis généré à partir de la relation binaire définie ci-dessus est non décomposable par la décomposition sous-directe. Nous appliquons les relations flèches afin d'obtenir un sous-contexte compatible. Soit O l'ensemble des classes feuilles de la hiérarchie de classes $\mathscr{H}$, A l'en-

semble des attributs (attributs formels) et I la relation binaire qui associe O à A. Comme les classes feuilles possèdent tous les attributs de la hiérarchie à cause de la définition de la relation binaire I, notre preuve est restreinte aux concepts feuilles.

Nous présentons une hiérarchie de classes $\mathscr{H}_i$, une hiérarchie qui possède i niveaux d'héritage dans sa structure. Par exemple, $\mathscr{H}_1$ a un seul niveau d'héritage entre les classes. R est la racine de la hiérarchie. Notre formalisme est le suivant :
$\forall$ classe $c \in \mathscr{H}_1$ et $c \neq R$, c hérite de R. Pour le cas de la hiérarchie $\mathscr{H}_2 = \mathscr{H}_1 \cup \mathscr{K}_1$ où $\mathscr{K}_1$ est l'ensemble des classes $\{c_i\}_{0 \leq i \leq n}$ tel que $\exists b \in \mathscr{K}_1, a \in \mathscr{H}_1$ avec b le descendant direct de a. Les classes de niveau i sont des descendants directs des classes de niveau *(i-1)*. Soit $\mathscr{L}_i$ le treillis de concepts de la hiérarchie $\mathscr{H}_i$. Supposons que nous pouvons décomposer une hiérarchie $\mathscr{H}_{n-1}$, et démontrons que nous pouvons le faire pour la hiérarchie $\mathscr{H}_n$.

En générant le treillis de $\mathscr{H}_n$, le concept attribut μa (pour chaque attribut a) est au-dessous du concept maximal du treillis (concept top). Cela signifie que $\mu a' = \{o/o \in O\}$ est en relation flèche descendante avec tous les attributs qui ne sont pas dans μa. Dualement, $\gamma o' = \{a/a \in A\}$ est en relation flèche ascendante avec toutes les classes qui ne sont pas dans γo. Le concept objet γo de chaque objet o est directement au-dessus du concept minimal du treillis (concept bottom). Cela signifie que tous les objets formels (les classes) et tous les attributs formels (attributs) de la table de contexte sont en liaison par les relations flèches. Nous obtenons que chaque classe de l'ensemble des classes joint tous les attributs :
Pour $o, a \in O \cup A, \exists (o_i, a_i)_{0 \leq i \leq k}$ avec $k \leq n$:
$o \swarrow\!\nearrow a_0 \swarrow\!\nearrow o_0 .... \swarrow\!\nearrow o_k \swarrow\!\nearrow a$.

Toutes les classes et les attributs sont en liaison. Comme la projection est du côté des attributs, il n'existe pas de sous-contexte compatible différent du contexte global qui regroupe tous les attributs. Le seul sous-contexte compatible (H, N) qui existe est le contexte global (O, A) et le treillis facteur n'est que le treillis global initial. Par suite, il n'existe pas de sous-contexte compatible pouvant être utilisé comme projection pour pouvoir décomposer le treillis en treillis facteurs. Nous confirmons que le treillis de concepts est non-décomposable.

Comme application de telle décomposition en utilisant les relations flèches sur l'exemple de la figure 6.5, nous obtenons la figure 6.7. Il n'est pas possible d'identifier un sous-contexte compatible puisque tous les attributs sont en relation.

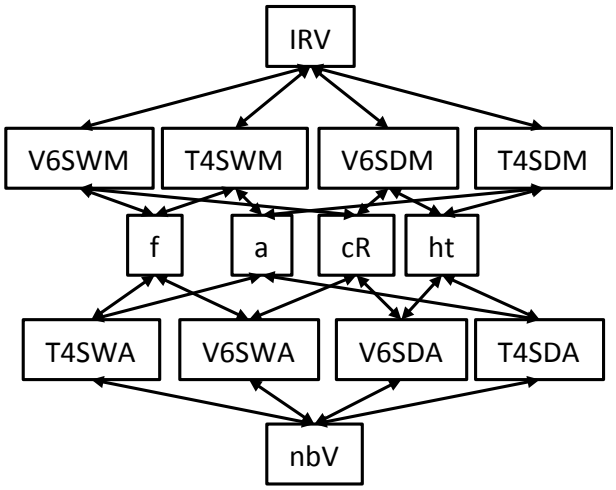


Figure 6.7 – Application des relation flèches sur l'exemple de la figure 6.5

Dans ce qui suit, nous allons décrire notre deuxième approche. Nous débutons par illustrer cette dernière à travers des exemples. Nous présentons et formalisons ensuite notre nouvelle manière de définir la relation binaire pour la génération de la table de contexte ainsi que la description de notre algorithme d'identification. Elle nous permet d'identifier les cas des occurrences multiples indépendamment des structures de classes sous-jacentes, y compris le cas parfait nommé multiplication d'états.

6.2 L'intuition derrière notre relation binaire

Une fonctionnalité peut se manifester dans l'occurrence d'un patron structurel de données/méthodes dans différents endroits d'une hiérarchie. Cet état des choses est illustré par les exemples de la figure 5.1 et la figure 6.8. D'une manière générale, la fonctionnalité est un ensemble de méthodes qui sont répartis sur un ensemble de classes. Nous estimons donc qu'il peut y avoir différentes répartitions concrètes.

La figure 6.8 illustre un exemple de cas de multiplication d'états. Une fonctionnalité est reconnue par l'occurrence d'un *patron* constitué de données/méthodes liés d'une manière hiérarchique dans différents endroits de la hiérarchie. Un tel patron illustre un comportement particulier sur les classes auxquelles il est associé. Dans ce cas, la fonctionnalité représente ce que cela signifie à être la *ressource de production* : une ressource a des *capacités* et un *calendrier*. Elle se spécialise dans des ressources de chaîne de montage (des machines-outils et des opérateurs) et de transport (matériel roulant et conducteurs).

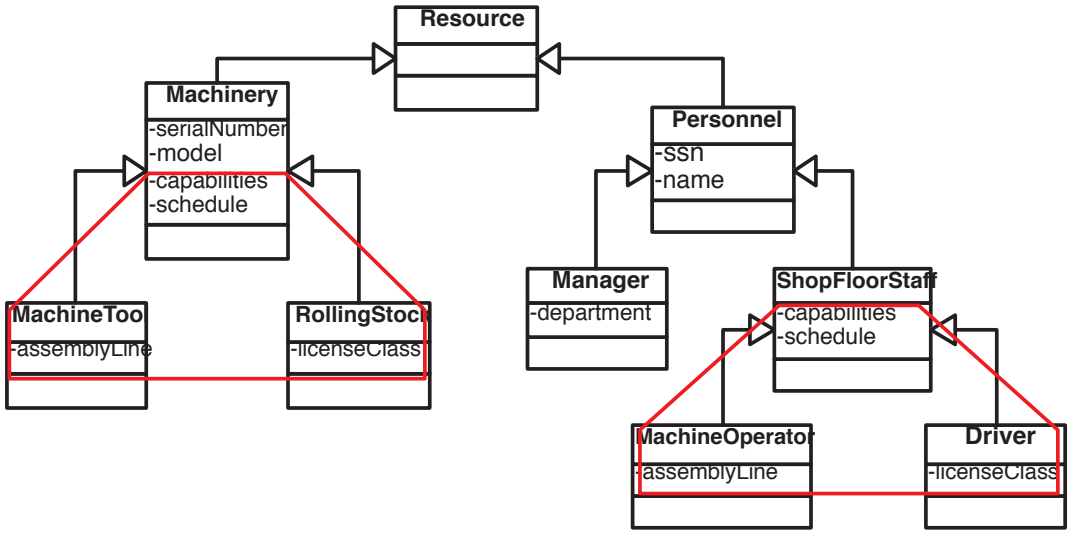


Figure 6.8 – Cas général d'une implémentation ad-hoc

Néanmoins, nous voulons utiliser l'AFC pour identifier les patrons récurrents dans un code légataire. Dans ce domaine de recherche, Arévalo [22, 23, 25] a aussi utilisé l'AFC pour identifier des patrons récurrents dans le code. Elle cherchait à identifier ce type de récurrence à travers les dépendances entre les classes (la relation d'héritage et la relation d'agrégat). Dans la figure 6.9, les patrons constitués des classes {A, B, C} et {D, E, F} sont équivalents de point de vue types de dépendances entre les classes. Ainsi, elle a pu utiliser la modélisation des entités du logiciel comme composants de l'AFC et la performance de l'AFC pour le regroupement de données afin de détecter des dépendances implicites, des patrons de conception et des défauts dans le système [21].

Peut-on utiliser le même principe, utiliser la modélisation des entités du logiciel comme composants de l'AFC et la performance de l'AFC pour le regroupement de données, afin d'identifier nos patrons ou ensembles de méthodes récurrents dans le code ?

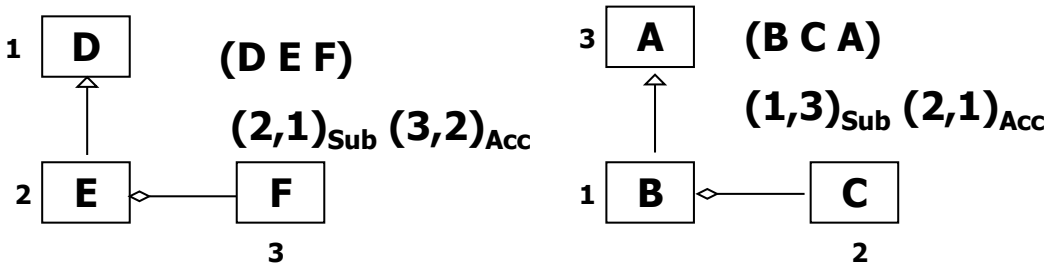


Figure 6.9 – Exemple de patrons récurrents de classes [24]

Notre but est de regrouper des "fragments de hiérarchie" basés sur un même ensemble de membres partagé. Nous avons essayé d'identifier ce type de fragments avec la relation binaire standard (pour chaque classe, associer ses attributs associés). Nous avons appliqué la relation binaire standard sur la hiérarchie de classes de la figure 6.8. Elle identifie les classes ayant le même ensemble de propriétés et manifestement ne permet pas la détection d'un patron récurrent de hiérarchie, c'est-à-dire de plusieurs classes (voir figure 6.10). De même, nous n'avons pu identifier notre patron récurrent avec la relation binaire "hérité de" telle que pour chaque classe, nous associons ses propres attributs et les attributs hérités (voir figure 6.11). La structure hiérarchique du patron rend l'identification difficile sur le treillis. Nous avons donc besoin de définir une nouvelle relation d'incidence entre les classes et les membres. Celle-ci devrait permettre de regrouper plusieurs classes en une seule entité indépendamment de la répartition des attributs. L'idée est d'avoir un objet formel qui regroupe plusieurs membres dispersés et appartenant à plusieurs classes. Idéalement, une occurrence doit être représentée par une classe sans pour autant correspondre à une seule classe. C'est une représentation d'une arborescence représentée par la classe racine qui couvre l'ensemble des classes de l'occurrence. Aussi, il faut avoir un seul concept qui correspondra à l'occurrence pour ne pas perdre l'avantage de l'AFC. C'est une approche différente des autres manières standards pour définir la relation binaire.

B_Standard

A	B	C	D	E	F	G	H	I	J
B_Standard	SerialNum...	model	capabilities	schedule	ssn	name	assemblyL...	licenseCla...	departement
Machinery	X	X	X	X	0	0	0	0	0
Personnel	0	0	0	0	X	X	0	0	0
MachineTool	0	0	0	0	0	0	X	0	0
RollingStock	0	0	0	0	0	0	0	X	0
Manager	0	0	0	0	0	0	0	0	X
ShopFloor...	0	0	X	X	0	0	0	0	0
MachineOp...	0	0	0	0	0	0	X	0	0
Driver	0	0	0	0	0	0	0	X	0

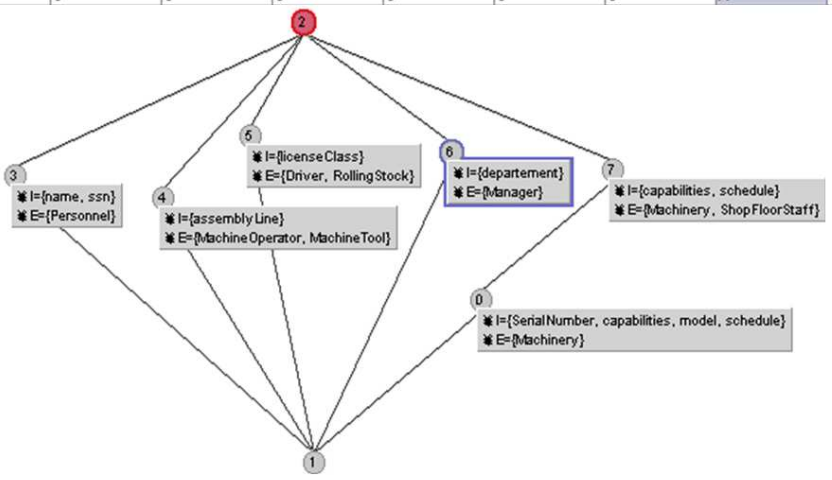


Figure 6.10 – Génération de treillis à partir de la relation binaire standard

B_Heritage

A	B	C	D	E	F	G	H	I	J
B_Heritage	SerialNum...	model	capabilities	schedule	ssn	name	assemblyL...	licenseCla...	departement
Machinery	X	X	X	X	0	0	0	0	0
Personnel	0	0	0	0	X	X	0	0	0
MachineTool	X	X	X	X	0	0	X	0	0
RollingStock	X	X	X	X	0	0	0	X	0
Manager	0	0	0	0	X	X	0	0	X
ShopFloor...	0	0	X	X	X	X	0	0	0
MachineOp...	0	0	X	X	X	X	X	0	0
Driver	0	0	X	X	X	X	0	X	0

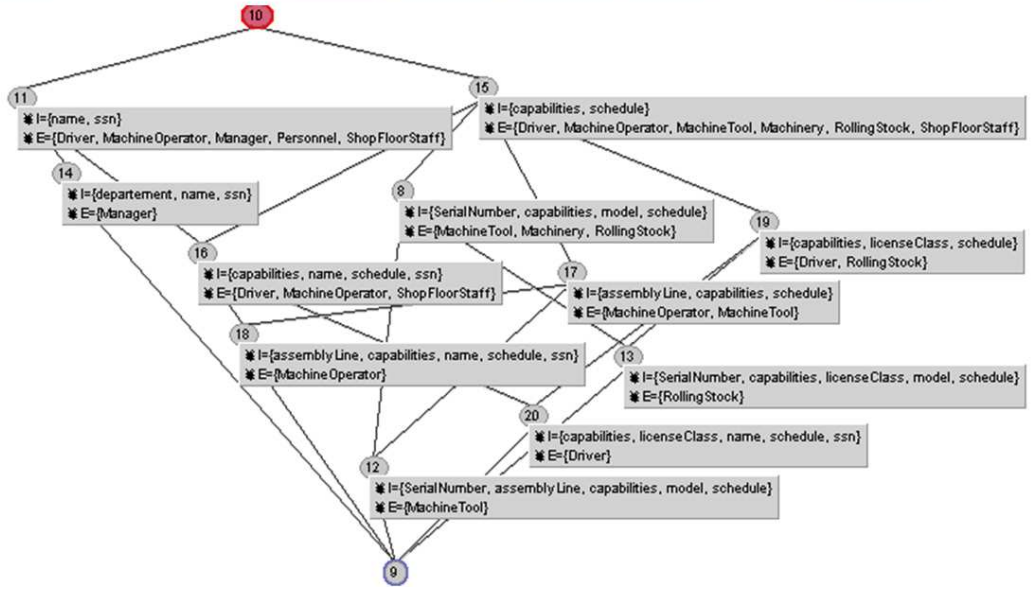


Figure 6.11 – Génération de treillis à partir de la relation binaire associée à l'héritage

C'est pourquoi, nous avons défini notre relation binaire comme suit : pour chaque classe *C* de la hiérarchie, associer tous les attributs qui apparaissent dans la sous-hiérarchie de racine cette même classe *C*, indépendamment de la position relative. Est ce que cela signifie pour autant que la structure n'est pas importante ? Pas nécessairement. Cependant, nous estimons que le même ensemble d'attributs doit pouvoir apparaître à plusieurs endroits de la hiérarchie avec une répartition différente. Par exemple dans la figure 6.13, les attributs *assemblyLine* et *licenseClass* qui sont dans les classes "descendantes" des classes ayant les attributs *{capabilities, schedule}*, peuvent apparaître sous forme d'une autre répartition à d'autres endroits. En appliquant notre nouvelle relation binaire sur l'exemple de la figure 6.8, nous obtenons le treillis associé illustré par la figure 6.12. Le nombre de concepts est nettement moins élevé que celui de la figure 6.11 et nous pouvons facilement noter que le concept '23' indique qu'il existe une redondance de l'ensemble d'attributs *{capabilities, schedule, assemblyLine, licenseClass}* pour les hiérarchies de classes ayant comme racines *{Machinery, Personnel, ShopFloorStaff}*.

D'une manière générale, la figure 6.13 montre le type de relation d'incidence que nous allons appliquer. Les entités sont sous forme des fragments de hiérarchies, dans ce cas précis, une classe X avec toutes ses sous-classes. Les propriétés sont l'ensemble de membres de données : *capabilities* et *schedule* dans la classe racine, *assemblyLine* et *licenseClass* dans les sous-classes. Cependant, un tel codage est complexe et restrictif. Pour un fragment de hiérarchie donné - par exemple, la classe *Machinery* avec les sous-classes *MachineTool* et *RollingStock* - le nombre d'ensembles de patrons d'attributs est exponentiel au nombre d'attributs des classes du fragment. Par exemple, soient l, m et n les nombres d'attributs des classes *Machinery*, *MachineTool* et *RollingStock* respectivement. Il y a $(2^l - 1)$ d'ensembles non vides d'attributs pour *Machinery*, et $(2^m - 1)$ d'ensembles non vide d'attributs pour *MachineTool* et ainsi de suite. Alors, il y a $(2^l - 1) \times (2^m - 1) \times (2^n - 1)$ patrons d'attributs possibles. Par ailleurs, la condition que les attributs *assemblyLine* et *licenseClass* soient dans les sous-classes immédiates de la classe racine n'est pas nécessairement une condition stricte.

B_New

A	B	C	D	E	F	G	H	I	J
B_New	SerialNum...	model	capabilities	schedule	ssn	name	assemblyL...	licenseCla...	departement
Machinery	X	X	X	X	0	0	X	X	0
Personnel	0	0	X	X	X	X	X	X	X
MachineTool	0	0	0	0	0	0	X	0	0
RollingStock	0	0	0	0	0	0	0	X	0
Manager	0	0	0	0	0	0	0	0	X
ShopFloor...	0	0	X	X	0	0	X	X	0
MachineOp...	0	0	0	0	0	0	X	0	0
Driver	0	0	0	0	0	0	0	X	0

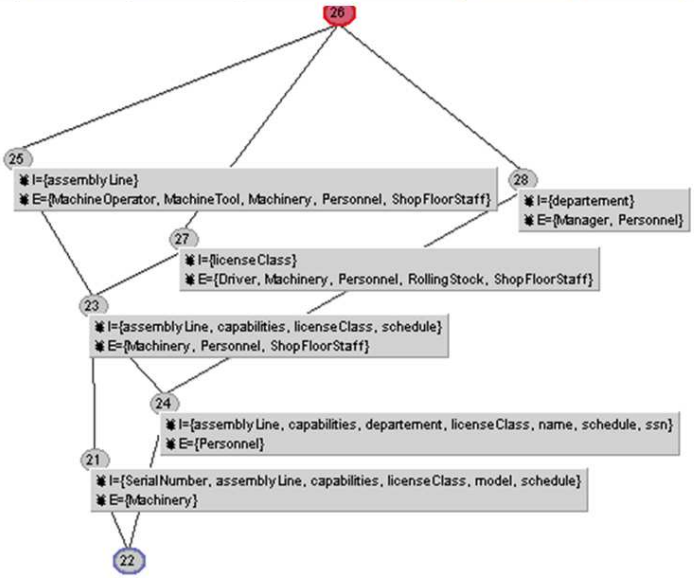


Figure 6.12 – Génération de treillis à partir de notre nouvelle relation binaire

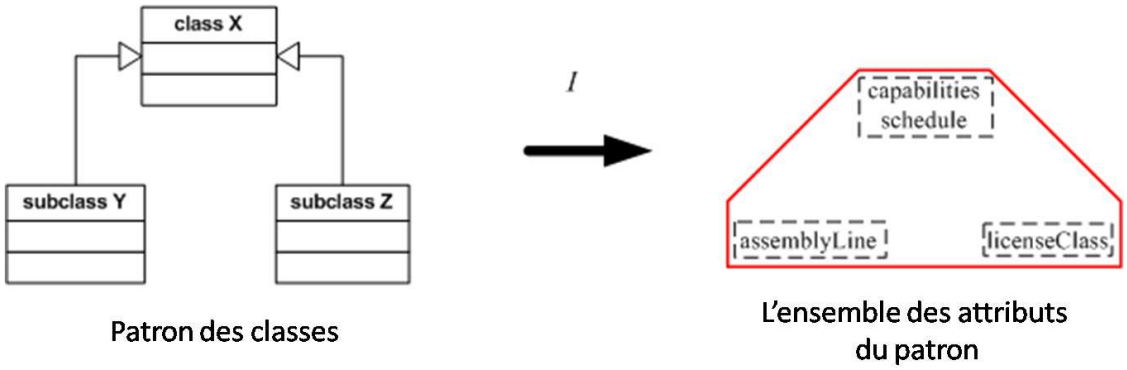


Figure 6.13 – Exemple de la relation d'incidence utilisée basée sur la figure 6.8

6.3 Formalisation de la représentation pour les occurrences

Définition 24 (Occurrence)**.** *Une occurrence O d'une fonctionnalité est définie par la paire* $[A,N]$*, avec A un ensemble de classes et N l'ensemble des membres associés. Chaque classe de A contribue un membre de N qu'aucune autre contribue.*

Une occurrence multiple est représentée par deux ensembles de classes, telle que $O_1 = [A_1,N]$ et $O_2 = [A_2,N]$ avec $A_1 \neq A_2$. Les membres d'une occurrence sont présentes dans une ou plusieurs classes. Comme l'espace de recherche sous-jacent peut être exponentiellement grand, il est préférable que les classes ne soient pas n'importe où dans la hiérarchie. En effet, si nous considérons tous les cas, le nombre d'objets potentiels serait combinatoire (2^n). Pour restreindre l'espace de recherche, nous nous concentrons sur les occurrences *compactes*.

Dans ce qui suit, nous formalisons notre manière intuitive de caractériser les occurrences. Soient une hiérarchie de classes notée $\mathscr{H} = \langle C, \leq \rangle$ avec la classe universelle $\top$, un ensemble de membres M et une relation d'incidence $I \subseteq C \times M$ telle que cIm ou $(c,m) \in I$ ssi la classe c définit la méthode m. Dans notre approche, deux ensembles ayant des méthodes communes correspondent à deux ensembles ayant des signatures de méthodes identiques. Une *fonctionnalité* est un ensemble de ces méthodes dans N avec $N \subseteq M$.

Définition 25 (Occurrence valide)**.** *: une occurrence valide (ou simplement occurrence) est défini par une paire* (A,N) *où l'ensemble des classes* $A \subseteq C$ *telle que :*

- *L'ensemble des membres A couvre N :* $N \subseteq \{m | \exists c \in A, cIm\}$*,*
- *Aucune classe n'est redondante dans A puisqu'elle contribue au moins un membre de N :* $\forall c \in A, (\exists m \in N : \forall \bar{c} \in A, (\bar{c},m) \in I \Rightarrow \bar{c} = c)$*.*

Par exemple dans la figure 6.14, $(\{c1,c5\},\{m2,m3\})$ et $(\{c1,c4\},\{m2,m3\})$ sont des occurrences valides de $\{m2,m3\}$ alors que $(\{c1,c3,c5\},\{m2,m3\})$ ne l'est pas puisque $c1$ et $c3$ ont comme seule contribution le même membre $m2()$. Évidemment, nous chercherons à détecter des candidats ayant au moins deux occurrences.

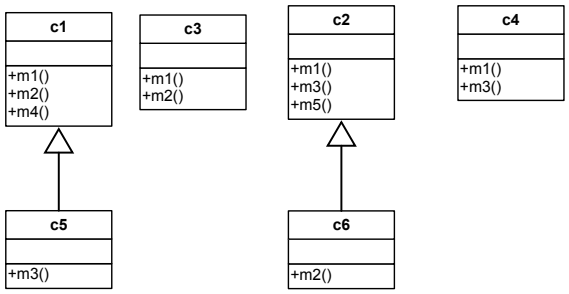


Figure 6.14 – Exemple de diagramme de classes

Il est à noter que la propriété ci-dessus est seulement une condition nécessaire pour notre approche. Chaque occurrence est identifiée avec une sous-hiérarchie distincte. Chaque sous-hiérarchie doit être caractérisée par un petit nombre de classes. Or toutes les classes au-dessus de la sous-hiérarchie, sont des majorants de celle-ci.

Idéalement, l'objectif est de donner à l'utilisateur des occurrences valides ayant un ensemble minimal de classes. Une manière directe de le faire est d'avoir un objet formel qui représente une occurrence. L'idée est de représenter chaque occurrence par un objet formel. Comme le nombre d'objets formels peut être grand, pour le diminuer, il faut restreindre les exigences sur l'ensemble des classes pour que l'ensemble soit restreint. Alors qu'il faut prendre tous l'ensemble des membres pour la complétude. Ainsi, nous avons pensé à éliminer certaines classes pour avoir un ensemble de classes restreint avec un ensemble de méthodes. Nous introduisons la notion de racines (*roots*), pour que chaque sous-hiérarchie soit caractérisée de manière minimale. Chaque occurrence candidate sera identifiée par une seule classe et une seule sous-hiérarchie. Dans la figure 6.15), l'occurrence représentée par les sous-hiérarchies A et B sera identifiée par les classes racines C_1, C_2.

Définition 26 (Roots)**.** *Une représentation canonique d'une occurrence appelée roots est définie comme les prédécesseurs minimaux communs de toutes les classes dans l'occurrence :*

- $roots(A,N) = \min(\{c | \forall \bar{c} \in A, \bar{c} \leq c\})$.

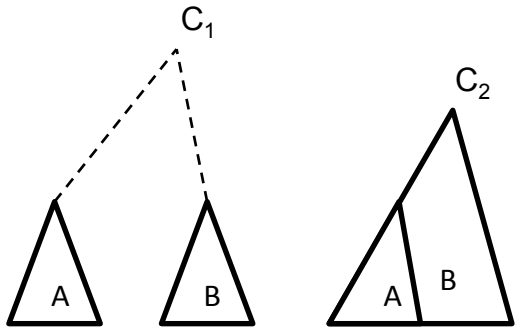


Figure 6.15 – Représentation d'une occurrence, composée des sous-hiérarchies A et B, par deux classes racines correspondantes

Par exemple (voir figure 6.14), $roots(\{c1,c5\},\{m2,m3\}) = \{c1\}$ et $roots(\{c1,c4\},\{m2,m3\}) = \{\top\}$ ($\top$ est la classe universel, *Object* en Java). Les occurrences *non-triviaux* sont celles qui ont leurs racines différentes de $\top$: $\top \notin roots(A,N)$ et deux occurrences $[A_1,N]$ et $[A_2,N]$ sont *indépendantes* si $roots([A_1,N]) \neq roots([A_2,N])$.

Plus loin, nous ciblons les occurrences *complètes* $[A,N]$, i.e. où A représente une sous-hiérarchie de $\mathscr{H}$ et N l'union de **tous** les membres associés à A. A est évidemment enraciné dans la classe unique $c \in roots([A,N])$, i.e. $c \in A$. Notons que les racines d'une occurrence peuvent appartenir, mais pas nécessairement à son ensemble de classes.

Définition 27 (Occurrence compacte)**.** *Une occurrence* (A,N) *est compacte si aucune autre occurrence de la même fonctionnalité N existe, soit* (B,N)*, tel que :*

- *Les ensembles de classes soient non disjoints :* $B \cap A \neq \emptyset$*,*
- *les racines de* (B,N) *soient inférieures à celle de* (A,N) *:* $\forall c \in roots(B,N), \exists \overline{c} \in roots(A,N) : c \leq \overline{c}$.

La deuxième condition signifie que la racine de l'occurrence compacte (A,N) est la plus petite racine de toutes les racines des autres occurrences de la même fonctionnalité, ayant au moins une classe dans A. Par exemple, $(\{c1,c4\},\{m2,m3\})$ n'est pas une occurrence compacte à cause de $(\{c1,c5\},\{m2,m3\})$. Chaque classe contribue au moins à une méthode qui lui est exclusive de N.

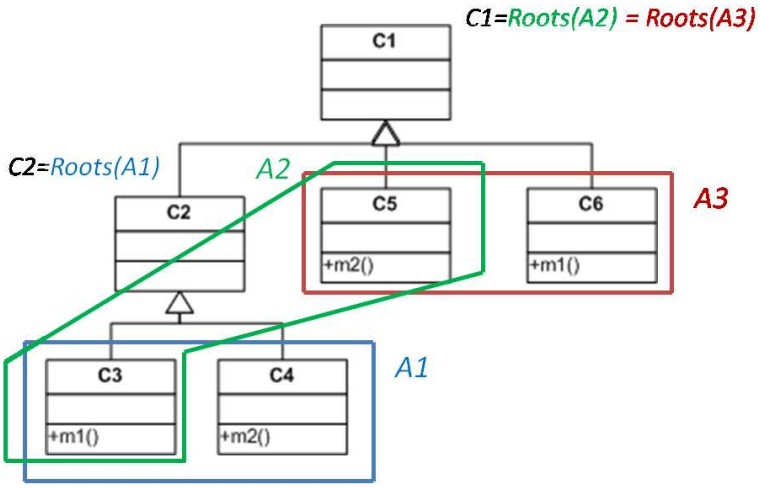


Figure 6.16 – Exemple de racines incomparables

Le problème se pose dans le cas où les racines des occurrences sont incomparables (exemple de la figure 6.16). Alors faut-il considérer tous les ensembles de N ? À cause des combinaisons des sous-ensembles de N, nous jugeons suffisant de prendre les ensembles maximaux qui englobent N et trouver les occurrences de ces ensembles. Pour qu'un ensemble de classes A puisse être une occurrence, il faut prendre tous ses membres de N. Nous considérons les sous-hiérarchies qui représentent cet ensemble A. Elles doivent être le plus petit ensemble de classes fermées qui recouvre ces sous-hiérarchies, ce qui mène à dire des sous-hiérarchies compactes. Techniquement, nous ignorons les sous-occurrences (B,Q) de (A,N) avec $B \subseteq A$ et $Q \subset N$.

Comme mentionné auparavant pour l'identification de ce type d'occurrences, nous allons utiliser l'AFC grâce à sa factorisation maximale et sa fermeture transitive. Nous proposons d'identifier les présences d'occurrences avec une super classe minimale commune[1] de toutes les classes contribuant à l'occurrence. Les entités maintenant vont correspondre aux sous-hiérarchies, représentées par leurs classes *racines*. En effet, cela signifie que nous ne nous soucions pas de la forme de la structure au-dessous de la racine. La figure 6.17 illustre notre nouvelle relation d'incidence.

Considérons encore l'exemple de la figure 6.8, et plus spécifiquement la branche

[1] évidemment, avec un héritage simple, l'ensemble de telles classes est un élément

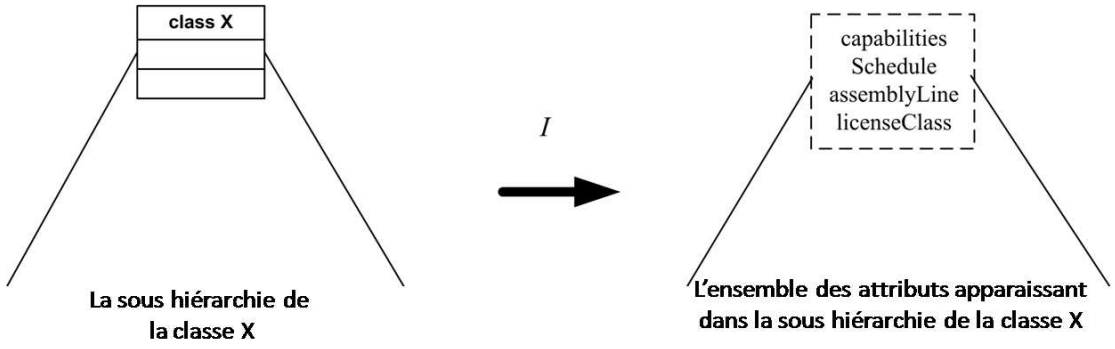


Figure 6.17 – Contexte simplifié basée sur la figure 6.8

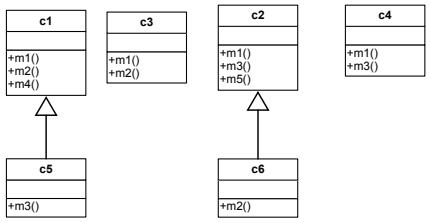
droite de la hiérarchie. Si nous ignorons la branche gauche de la sous-hiérarchie de racine *Machinery*, avec notre relation d'incidence, nous remarquons le concept où l'extent a trois classes *{Resource, Personnel, ShopFloorStaff}* et l'intent possède les attributs *{capabilities, schedule, assemblyLine,licenseClass}*. Nous ne pouvons dire que l'ensemble de ces attributs se trouve à plusieurs endroits, la raison étant que les trois occurrences correspondent en fait à trois sous-hiérarchies emboîtées (voir figure 6.20).

Nous allons présenter dans la partie qui suit la traduction typique d'une hiérarchie de classes dans les termes de l'AFC, ainsi que le problème d'identification des fonctionnalités avant de présenter la localisation des méthodes associées dans le code. Ensuite, nous allons détailler notre relation binaire ainsi que notre algorithme d'identification [50, 51].

6.4 Relation binaire et algorithme

Dans l'AFC, les concepts apparaissent à partir d'un contexte (formel) $\mathscr{K} = (O, A, I)$ où O est un ensemble d'entités (des objets formels), A un ensemble de propriétés (des attributs formels) et I (la relation d'incidence) qui associe O à A : $(o, a) \in I$ quand l'entité o possède la propriété a. La figure 6.18 (dans la partie à droite) fournit un exemple d'un contexte où les entités sont des classes et les propriétés sont les membres des classes. La relation binaire I dans ce contexte n'est pas une relation binaire standard *avoir-membre* entre des classes et des membres, mais c'est un codage spécifique des hiérarchies de classes que nous discutons ci-dessous comme dans le diagramme de classes de la fi-

gure 6.18 (voir la partie à gauche). Dans le contexte, les paires qui appartiennent à la relation d'incidence sont dénotées par X et celles qui n'en font pas partie par 0. Également, la relation d'incidence entre une classe et un membre devient l'opposé de la relation d'héritage : une classe est maintenant rapprochée de tous les membres qui sont définis par la classe elle-même ou par une de ses sous-classes. La figure 6.18 fournit un exemple d'ensembles de classes et son codage dans un contexte, la classe racine universelle a été omise tant dans le modèle que dans le contexte en raison de son impact nul sur la structure de treillis. Le *treillis de concept* $\mathscr{L}$ du contexte $\mathscr{K}$, est présenté dans la figure 6.19.



Funct...	m1	m2	m3	m4	m5
c1	X	X	X	X	0
c2	X	X	X	0	X
c3	X	X	0	0	0
c4	X	0	X	0	0
c5	0	0	X	0	0
c6	0	X	0	0	0

Figure 6.18 – **Gauche :** Les Classes initiales ; **Droite :** La table de contexte associée.

Un ensemble d'entités $X \subseteq O$ a une image dans A est défini par $X^I = \{a \in A \mid \forall o \in X,\ (o,a) \in I\}$. Dualement, chaque ensemble de propriétés $Y \subseteq A$ a une image dans O défini par $Y^I = \{o \in O \mid \forall a \in Y,\ (o,a) \in I\}$. Dans notre exemple, soit $X = \{c1, c4\}$, alors $X^I = \{m1, m3\}$, où $Y = \{m1, m3\}$, $Y^I = \{c1, c2, c4\}$. Un *concept* est une paire (X,Y) où $X \subseteq O, Y \subseteq A, X^I = Y$ et $Y^I = X$.

Dans la figure 6.19, le concept $\{\{c1, c2\}, \{m1, m2, m3\}\}$ spécialise $\{\{c1, c2, c4\}, \{m1, m3\}\}$. Les lignes / colonnes remplies exclusivement par X ou par 0 dans un contexte sont habituellement ignorés dans l'AFC puisqu'ils n'ont pas d'impact sur la structure. Grâce à la factorisation orientée objets [68], une interprétation directe du concept 5 du treillis de la figure 6.19, pourrait suggérer la conception d'une nouvelle classe (ou interface) ayant la déclaration de $m1()$ qui est partagée par les quatre classes. L'interprétation d'application de l'AFC pour une refactorisation de hiérarchies de classes [68] a l'avantage d'avoir une factorisation maximale de propriétés et une conformité d'héritage entre

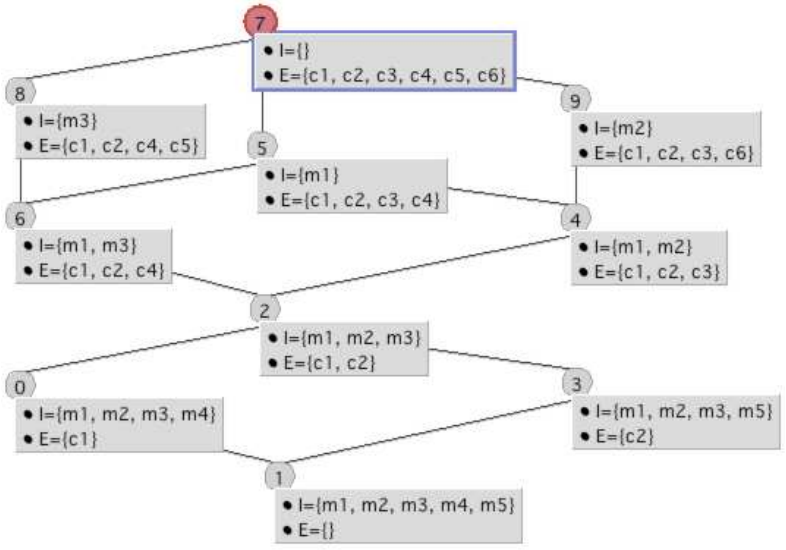


Figure 6.19 – Le treillis de concepts associé à la figure 6.18

ses membres.

Les intentions des concepts dans le treillis sont exactement les modèles fermés (c'est-à-dire, maximal pour l'ensemble des entités qui apparaissent) de l'ensemble de données défini par le contexte. Notre but est de détecter le maximum des membres partagés. Quoique la question demeure encore comment traduire une hiérarchie dans un contexte approprié.

6.4.1 Relation binaire

La nouvelle relation fonctionne dualement par rapport au mécanisme d'héritage. Comme son nom l'indique, la relation *anti-héritage J* d'une hiérarchie $\mathscr{H}$, avec O l'ensemble des classes et A l'ensemble des méthodes, est définie comme suit [50, 51] :

Définition 28 (Relation *anti-héritage*)**.** $J \subseteq O \times A$ *comme oJa ssi :*

1. oIa
2. *ou* $\exists \bar{o} \leq o$ *avec* $\bar{o}Ia$.

J est la relation d'incidence dans le contexte $\mathscr{K} = (O, A, J)$ et va être utilisée comme opérateur de dérivation (i.e. nous écrirons o^J). Dans la figure 6.18, la relation d'incidence

dans le contexte (figure à droite) est exactement la relation anti-héritage entre les classes du diagramme (figure à gauche). Par exemple, $c1$ possède $m3()$ qui est défini dans une de ses sous-classes ($c5$).

Par exemple, dans une occurrence arbitraire, $[A,N]$, $o \in roots([A,N])$ implique que $N \subseteq o^J$ ou dans un cas plus complet $N = o^J$. Le contexte $\mathscr{K}$ comprend tous les objets correspondants aux occurrences complètes de la hiérarchie.

Les fonctionnalités ciblées correspondent à regrouper au moins deux occurrences indépendantes [2] du même ensemble N. A cause de la relation d'inclusion entre les intents des concepts, les ensembles maximaux N sont facilement détectés. Un filtrage manuel des fonctionnalités candidates est exigé.

Chaque fonctionnalité candidate potentielle N force le concept sous-jacent (X,Y) du contexte $\mathscr{K}$ $(Y = N)$ de suivre un patron structurel distinct dans le treillis $\mathscr{L}$ associé à $\mathscr{K}$. Les racines de toutes les occurrences de N dans $\mathscr{H}$ appartiennent à X.

Or, il ne faut pas que les classes encombrent l'extent X de notre concept. Il faut garder uniquement les classes utiles qui représentent les occurrences de N. Pour cela, il faut réduire le nombre des objets formels en ayant un maximum de propriétés formelles. Ainsi une autre contrainte s'impose : une occurrence doit être maximalement étendue du point de vue propriétés et minimalement restreinte du point de vue objets formels (classes). Pour avoir une représentation simple et facile à interpréter, X doit être composé uniquement des racines minimales qui seront notées par $\min(X)$. Comme nous exigeons d'avoir au moins deux occurrences indépendantes, il est nécessaire que $|\min(X)| \geq 2$. Nous avons aussi besoin d'éviter l'identification des sous-candidats correspondants aux sous-parties récurrentes des fonctionnalités. Par exemple, il est inutile de considérer l'occurrence de *assemblyLine* correspondant aux classes *MachineTool* et *MachineOperator* dans la figure 6.8, puisqu'une telle occurrence est une partie de l'occurrence d'une même fonctionnalité plus large.

Autre caractéristique, les concepts candidats ne doivent pas faire partie d'une même chaîne dans le treillis en suivant l'inclusion des ensembles entre les parties de la même occurrence (cela dû à la monotonie de $\_^J$ associée à $\leq$). En termes mathématiques, le

[2] mais non nécessairement complètes

concept (X,Y) ne sera pas considéré s'il possède un prédécesseur direct, (X_1,Y_1), avec le même nombre de racines minimaux dans son extent ($|\min(X)| = |\min(X_1)|$).

Dans notre exemple de la figure 6.19, les concepts 2, 4, 5, et 6 satisfont les conditions discutées ci-dessus. Par exemple, le concept 5 a quatre classes dans son extent, toutes les classes sont des classes minimales, où deux de ses prédécesseurs immédiats, 4 et 6, ont aussi trois classes minimales. Par contre, le concept 8 ne vérifie pas la caractéristique des classes minimales puisque parmi les quatre classes dans son extent, trois seulement sont minimales et il a la même cardinalité de l'extent de son prédécesseur immédiat, le concept 6. L'interprétation de cet exemple est qu'il existe deux fonctionnalités, $\{m1,m2\}$ et $\{m1,m3\}$, ayant trois occurrences indépendantes. La combinaison de ces deux fonctionnalités, $\{m1,m2,m3\}$, est une fonctionnalité ayant deux occurrences. Il est délicat de décider laquelle représente une fonctionnalité appropriée. Par suite, les trois ensembles sont considérés comme des candidats qui peuvent représenter différentes fonctionnalités.

Avec cette nouvelle relation d'incidence, les concepts consistent des paires (A,N) où pour chaque classe $o \in A$, la sous-hiérarchie de racine o a tous les attributs dans N.

Propriété 2 (Concepts maximaux)**.** *Les deux ensembles A et N du concept* (A,N) *sont maximaux tels que :*

1. *il n'existe pas de classe* $\bar{o}$ *à l'extérieur de A qui a tous les attributs dans N*
2. *et il n'existe pas d'attribut à l'extérieur de N commun à toutes les classes (ou sous-hiérarchies de racines) dans N.*

Par suite, comme première identification de fonctionnalité, chaque concept (A,N) où A contient plus d'une classe peut suggérer que N représente une fonctionnalité. Or, chaque concept ayant plus d'une classe (sous-hiérarchie) dans son extent, ne représente pas automatiquement une fonctionnalité : les classes qui sont dans l'extent ne doivent pas être reliées hiérarchiquement (par héritage). En pratique, cela signifie, avant de considérer l'intent (l'ensemble N) d'un concept (A,N) comme une fonctionnalité potentielle, que nous devons considérer la cardinalité de A où A doit contenir uniquement les classes minimales en respectant l'ordre défini par la relation des sous-classes. Dans l'exemple

de la figure 6.20, l'extent *{Resource, Personnel, ShopFloorStaff}* a une seule classe minimale en respectant les relations des sous-classes, qui est la classe *ShopFloorStaff*. Ainsi, en se basant seulement sur la branche de droite, nous ne pouvons dire que l'ensemble *{capabilities, schedule, assemblyLine, licenseClass}* représente une fonctionnalité.

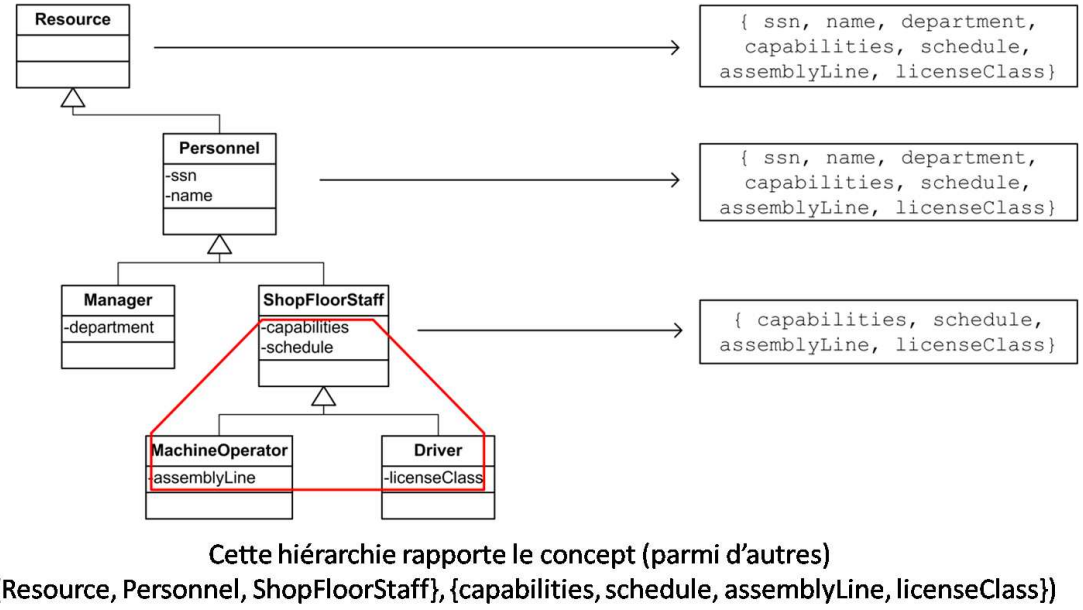


Figure 6.20 – Les trois occurrences de *{capabilities, schedule, assemblyLine, licenseClass}* ne sont pas indépendantes

Notons que si pour un ensemble d'attributs donné ayant deux ou plusieurs occurrences indépendantes, étant une fonctionnalité candidate, a fortiori, chaque sous-ensemble de cet ensemble d'attributs a au moins le même nombre d'occurrences indépendantes sinon plus. Certains de ces sous-ensembles peuvent eux aussi être qualifiés comme des fonctionnalités candidates. Comme à la figure 6.8, selon notre définition, les sous-ensembles (singletons dans ce cas) *{assemblyLine}* ou *{licenseClass}* ont déjà deux occurrences indépendantes (voir figure 6.21). Pour un nombre donné d'occurrences, nous nous intéressons aux fonctionnalités candidates ayant le nombre d'occurrence le plus large au niveau des attributs, i.e une fonctionnalité incluant le plus grand nombre d'attributs.

La figure 6.22 montre un extrait du treillis de concepts généré. Les trois concepts

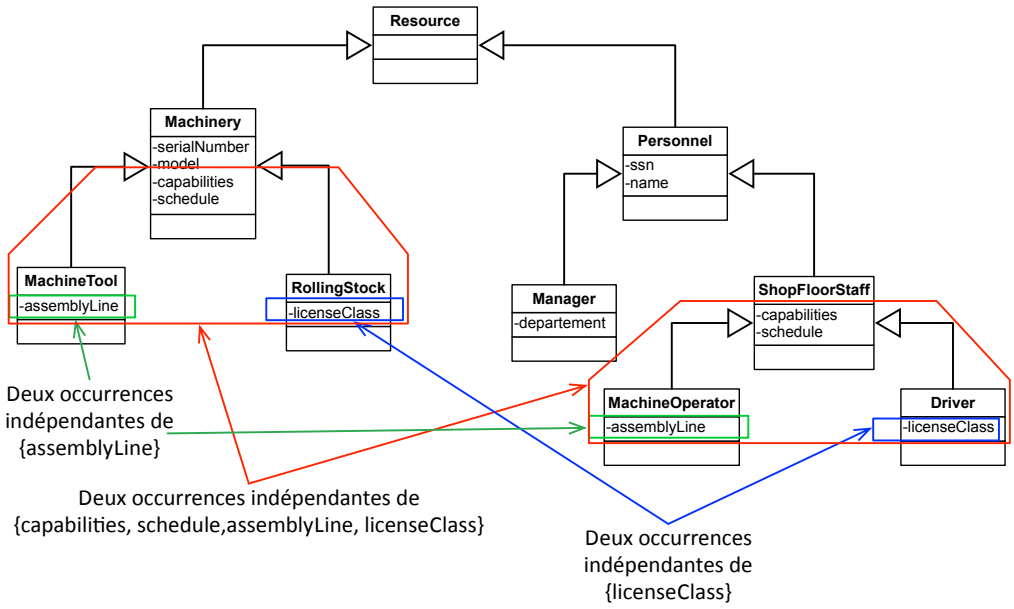


Figure 6.21 – Une fonctionnalité candidate peut induire plusieurs sous-fonctionnalités candidates

candidats ont deux occurrences indépendantes. Après la modification des concepts du treillis en éliminant les classes non minimales dans les extents, ces concepts candidats ont le même nombre de classes minimales (deux dans ce cas). Nous déduisons que *{assemblyLine}* et *{licenseClass}* correspondent effectivement à deux occurrences. Notons que si chacun des attributs *{assemblyLine}* ou *{licenseClass}* est localisé ailleurs, leurs extents correspondants vont avoir un plus grand nombre de classes minimales (trois ou plus), et vont être retenus comme une fonctionnalité potentielle.

6.4.2 Algorithme d'identification de fonctionnalités dans un treillis de Galois

Soit un programme P associé à une hiérarchie de classes $\mathscr{H} = \langle C, \leq \rangle$ avec un ensemble de membres M. Le contexte $\mathscr{K} = (C, M, J)$ est généré avec la relation d'incidence *anti-héritage* J. Pour extraire les fonctionnalités existantes dans P, nous analysons le treillis de concepts $\mathscr{L}$ généré à partir de la table de contexte $\mathscr{K}$ associée à la relation binaire J. Pour chaque concept (X, Y) de $\mathscr{L}$, les classes minimales, $\min(X)$ sont calculées dans l'extent.

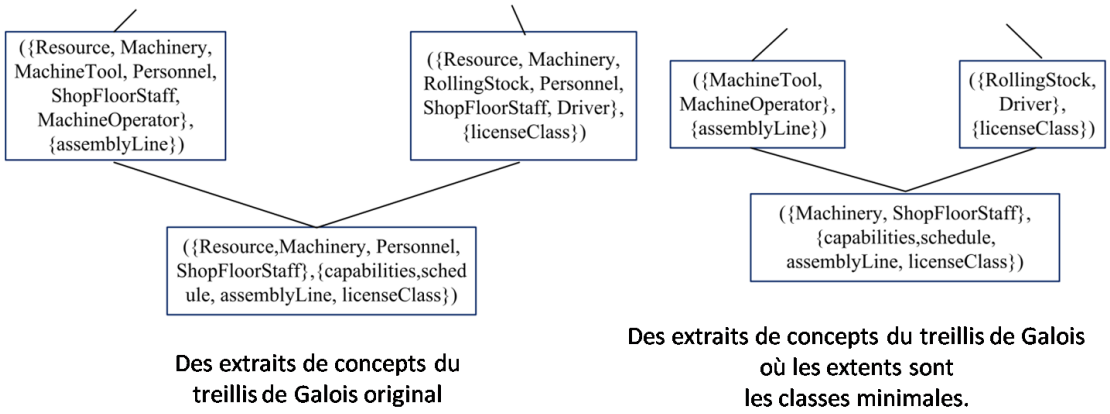


Figure 6.22 – Extrait du treillis de concepts de la figure 6.21

L'algorithme d'identification prend comme entrée le treillis de concepts $\mathscr{L}$, qui contient dans les extents de ses concepts uniquement les classes minimales, et produit comme sortie la liste des candidats potentiels à représenter des fonctionnalités (notée *CandFeatureList*). Pour chaque concept (X,Y), à partir du top du treillis ($\top_{\mathscr{L}}$), l'algorithme vérifie les conditions citées auparavant, i.e. avoir au moins deux classes dans $\min(X)$ et ne pas avoir un concept prédécesseur immédiat ayant la même cardinalité (même nombre) que son ensemble de classes minimales dans l'extent. Les concepts qui satisfont ces deux conditions sont ajoutés dans la liste des concepts candidats *CandFeatureList* pour une vérification manuelle ultérieure.

Input: concept lattice $\mathscr{L}$

Output: feature candidates *CandFeatureList*

ListConcept $\leftarrow$ children($\top_{\mathscr{L}}$)

CandFeatureList $\leftarrow \emptyset$

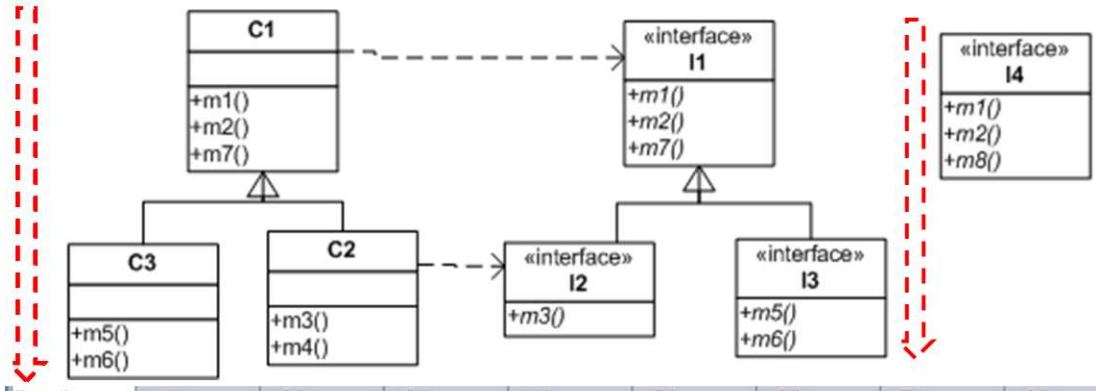
while *ListConcept* $\neq \emptyset$ **do**
    $(X,Y) \leftarrow$ extract(*ListConcept*)
    if $|\min(X)| > 1$ **then**
        add((X,Y), *CandFeatureList*)
        foreach $(\underline{X},\underline{Y}) \in$ *children*$((X,Y))$ **do**
            add($(\underline{X},\underline{Y})$, *ListConcept*)
            if $(|\min(\underline{X})| = |\min(X)|)$ **then**
                remove((X,Y), *CandFeatureList*)
        end
    end
end

6.5 Application de l'algorithme

La relation d'incidence prend en considération les classes et ses sous-classes ainsi que les interfaces et ses sous-types. Pour identifier la similarité, il faut tenir compte des APIs (interfaces) et des classes propres au code (voir figure 6.25). De la même manière que les classes minimales dans les concepts du treillis, nous calculons les interfaces minimales des hiérarchies d'interfaces.

Dans la figure 6.23, nous illustrons un exemple général avec deux hiérarchies pour mieux éclaircir le déroulement de l'algorithme, une hiérarchie de classes et une hiérarchie d'interfaces. A première vue, nous décelons les ensembles de méthodes communes entre interfaces et entre interfaces et classes. Comme la classe *C1* implémente l'interface *I1*, c'est normal d'avoir des méthodes communes. Aussi, des méthodes communes entre *C1*, *I1* et *I2*. En gardant juste les classes minimales et les interfaces minimales dans le treillis associé, la relation nous fait sortir facilement les concepts candidats (voir

figure 6.24).



Function	m1()	m2()	m3()	m4()	m5()	m6()	m7()	m8()
I1	X	X	X	0	X	X	X	0
I2	0	0	X	0	0	0	0	0
I3	0	0	0	0	X	X	0	0
I4	X	X	0	0	0	0	0	X
C1	X	X	X	X	X	X	X	0
C2	0	0	X	X	0	0	0	0
C3	0	0	0	0	X	X	0	0

Figure 6.23 – **Haut :**Comment la relation binaire est générée entre classes et entre interfaces ; **Bas :**Table de contexte associée

Aussi, grâce à la factorisation maximale du treillis de Galois, les concepts 13 et 15 dans la figure 6.24 (en-bas) regroupent clairement les méthodes communes entre une classe et une interface comme *(C3,I3),(m5(),m6())* et entre deux interfaces comme *(C1,I1,I4),(m1(),m2())* sans avoir une relation d'implémentation ou d'héritage entre elles.

La figure 6.25 fournit un autre exemple de diagramme de classes, plus réel, des différentes catégories des êtres vivants. Les relations existantes sont la relation d'héritage et la relation "implements" entre classes et interfaces. Les entités sont les classes et les interfaces, les propriétés sont les signatures de leurs méthodes. Chaque classe possède ses propres méthodes ainsi que les méthodes de ses sous-classes, et chaque interface possède ses propres méthodes ainsi que les méthodes de ses sous-interfaces. Par exemple dans notre cas, l'interface $Mammiferes\_T$, qui représente les mammifères terrestres, est en relation avec les méthodes $\{r(),c(),m(),a()\}$. La classe $Ursidés$ a le même en-

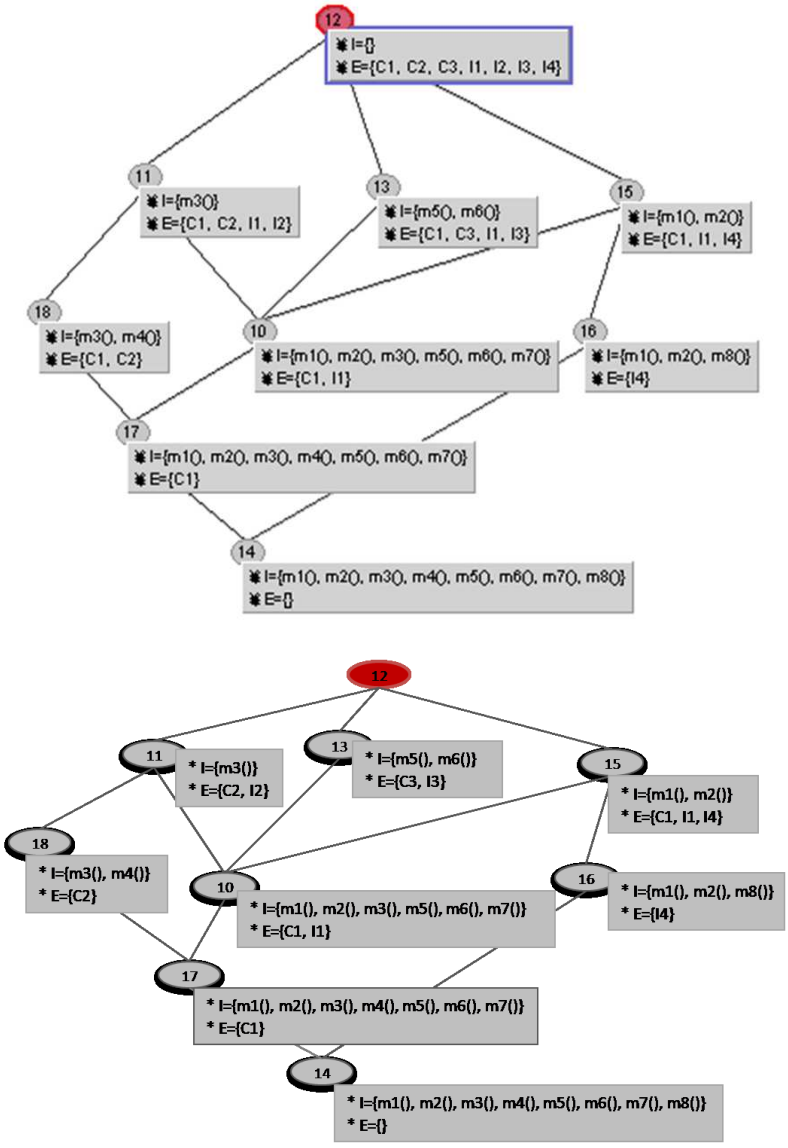
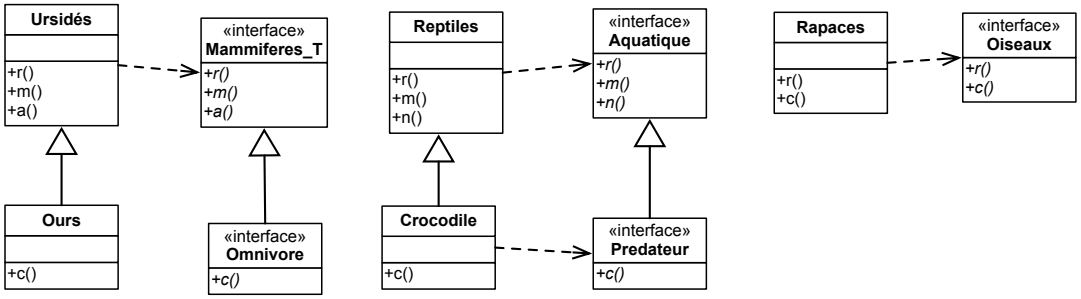


Figure 6.24 – **Haut :**Le treillis de concepts associé à la figure 6.23 ; **Bas :** Le treillis de concepts uniquement avec les classes et les interfaces minimales



r() : respirer, c() : chasser, m() : marcher, a() : allaiter, n() : nager()

Default Na...	r()	c()	a()	m()	n()
Ursides	X	X	X	X	0
Mammifere...	X	X	X	X	0
Ours	0	X	0	0	0
Omnivore	0	X	0	0	0
Reptile	X	X	0	X	X
Aquatique	X	X	0	X	X
Crocodile	0	X	0	0	0
Predateur	0	X	0	0	0
Rapace	X	X	0	0	0
Oiseaux	X	X	0	0	0

Figure 6.25 – **Haut :** Un exemple de diagramme de classes pour la relation d'incidence ; **Bas :** La table de contexte associée

semble de méthodes. De même l'interface *Aquatique* est en relation avec les méthodes *{r(),c(),m(),n()}*. La classe *Reptile* a aussi le même ensemble de méthodes.

La figure 6.26 illustre le treillis de concepts générés de la table de contexte de la figure 6.25. Remarquons que le concept '0' a comme intent *{r(), c(), m(), a()}* et comme extent *{Mammiferes_T, Ursidés}*. La cardinalité de celui-ci est deux et *Mammiferes_T* et *Ursidés* sont minimales. L'interface *Mammiferes_T* se trouve dans la partie *extent* du concept. Cela nous facilite la tâche d'interprétation au moment de l'analyse du code de la manière suivante : si la classe *Ursidés* implémente l'interface *Mammiferes_T*, nous marquons la fonctionnalité candidate comme une fonctionnalité déjà reconnue. Dans le cas contraire (pas de lien d'implémentation), elle est marquée comme un cas ad-hoc où nous suggérons au développeur d'avoir une relation *implements* entre la classe *Ursidés* et l'interface *Mammiferes_T*. Comme par exemple dans le cas entre la classe *Ours* et l'interface *Omnivore*. Après analyse du résultat, ce cas-ci sera considéré comme étant un cas ad-hoc. Nous remarquons de même que la méthode *r()* est commune à plusieurs interfaces et classes. Ce cas aussi est identifié comme étant un cas ad-hoc et il sera suggéré comme un cas à analyser.

6.6 Conclusion

Dans ce chapitre, nous avons proposé notre deuxième approche en utilisant les treillis de Galois. Elle présente une nouvelle manière pour définir la relation d'incidence afin d'identifier et localiser les fonctionnalités récurrentes des systèmes logiciels. Les treillis de Galois sont très connus par leur puissance de factorisation maximale. Avec notre relation binaire, nous arrivons à identifier rigoureusement les concepts candidats qui représentent des fonctionnalités récurrentes ou des cas à analyser pour améliorer la conception. Elle nous permet d'identifier certains défauts de conception et de mal factorisation, et nous permet aussi d'identifier des cas de délégation. Certains de ces cas sont identifiés dans le chapitre de validation.

La validation de nos deux approches sera présentée dans le chapitre suivant. La première approche se focalise sur l'identification des fonctionnalités issues de l'héritage et

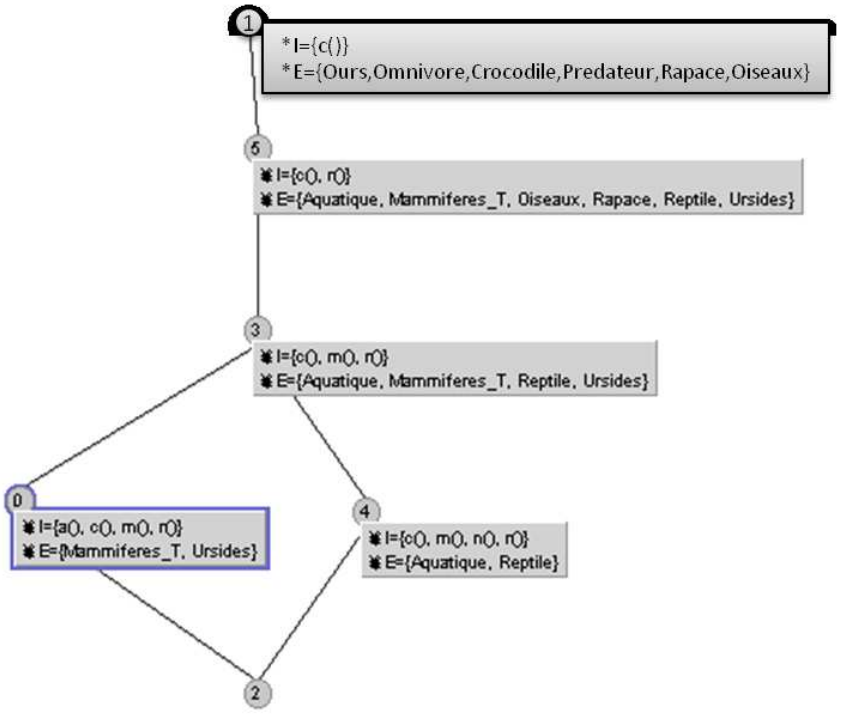


Figure 6.26 – Le treillis de concepts associé à la figure 6.25 avec classes minimales.

de la délégation. La deuxième approche identifie les fonctionnalités dans le cas de multiplication d'états. Nous allons appliquer nos différents algorithmes sur un ensemble de logiciels libres différents de point de vue taille, type et degré de qualité. Les résultats sont très prometteurs.

CHAPITRE 7

EXPÉRIMENTATION ET VALIDATION

Nous allons expérimenter nos différents algorithmes d'identification de fonctionnalités sur des logiciels libres implémentés en Java. Ils sont de différentes natures. Certains de ces logiciels sont supposés avoir une bonne conception et donc bien factorisés, comme JHotDraw, alors que pour d'autres logiciels, l'important est le fonctionnement et non la qualité de la conception.

La validation de nos algorithmes sur ces types de logiciels cherche à évaluer la pertinence de nos techniques pour la compréhension du code. Elle nous a montré que ces algorithmes peuvent aider les développeurs à comprendre la conception du logiciel, la mise à jour et nous facilitent la suggestion d'une meilleure factorisation.

Dans ce qui suit, nous débutons par la description des différents outils. Ensuite, nous présentons l'étude expérimentale sur l'identification des fonctionnalités issues de l'héritage multiple et de la délégation à travers un agrégat, et celle issue de la multiplication d'états. Chaque étude est présentée avec des tableaux résultats représentants les candidats associés aux différentes fonctionnalités pour chaque outil. Une analyse des résultats obtenus conclut ce chapitre.

7.1 Outils

Les gabarits des différents logiciels étudiés avec les informations sur ces applications sont fournies dans le tableau 7.I. Celles-ci ont différents niveaux de maturité, allant de la version 0.7.1 pour FreeMind, à la version 5.2 pour JHotDraw. Ici, le type d'application – domaine d'application– signifie que les applications n'ont pas la même qualité de conception.

Nous utilisons l'API JDT d'Eclipse pour analyser les codes légataires des logiciels dans le but d'extraire les signatures des méthodes des différentes classes (l'ensemble des données publiques (attributs) et les méthodes publiques) avec les types de relations

entre elles. Les types de relations sont les relations entre les classes (spécialisation), les relations entre interfaces (sous-typage), la relation *implements* entre les classes et les interfaces et la relation d'agrégation entre les classes (cas où une classe A a un membre de données de type classe B). Dans notre analyse, nous ignorons les signatures de méthodes qui proviennent de la librairie Java.

Outils	Fichiers sources	Lignes de Code	Classes et Interfaces	Méthodes
JHotDraw 5.2	160	9419	171	1229
JavaWebMail 0.7	111	10707	115	1079
JreversePro 1.4.1	83	9656	87	663
FreeMind 0.7.1	92	65490	198	4785
Lucene 1.4	160	15480	197	1270

Tableau 7.I – Liste des systèmes étudiés

7.1.1 JHotDraw

Développé par Erich Gamma et Thomas Eggenschwiler en se basant sur l'original écrit en Smalltalk par John Brant [30], JHotDraw représente un cadriciel (*framework*) d'interface graphique. Les objectifs principaux de JHotDraw ont été de servir comme un exercice ou un laboratoire de conception orienté objets. Le projet JHotDraw permet d'ajouter des fonctionnalités à la structure de base. Un bon nombre d'applications graphiques ont été développées en utilisant JHotDraw [8]. Nous utilisons la version 5.2 qui contient 160 fichiers sources et 171 classes et interfaces.

7.1.2 JrevsersePro

JreversePro est un programme Java pour la rétro-ingénierie (reverse engineering) du code Java compilé. C'est un décompilateur et désassembleur entièrement écrit en Java. Il permet de mieux gérer les classes, qui peuvent être désassemblées. Il prend comme entrées les fichiers .class (le résultat de la compilation des fichiers sources java), et peut produire une des trois structures, dépendamment des paramètres appelés :

1. classe d'association (constants pool) qui est un type dans la table de symboles dans la machine virtuelle Java standard ;
2. classe désassembleur ;
3. et classe de décompilation.

JreversePro est relativement petit avec 87 classes et interfaces et 9656 lignes de code. Il utilise juste l'API Java de base. Contrairement à JHotDraw qui a été amélioré par la contribution de plusieurs concepteurs, JreversePro est principalement le fruit des efforts de ses créateurs initiaux.

7.1.3 JavaWebMail

JavaWebMail est un autre logiciel libre visant à développer en Java standard une interface client de messagerie sous forme d'application web en utilisant les protocoles IMAP ou POP. Comme il est une application Java, il profite de la portabilité, de l'adaptabilité et d'une bonne performance (exécution). La version utilisée est JavaWebMail 0.7. Elle contient 115 classes et interfaces.

7.1.4 FreeMind

FreeMind permet de créer des cartes conceptuelles (Mind Map), des diagrammes représentant les connexions sémantiques entre différentes idées. FreeMind peut notamment aider à l'organisation des idées ou à la gestion de projets. Il permet de mettre en œuvre des textes sous forme d'une hiérarchie ou d'un plan pour avoir une meilleure présentation qu'un simple système (traitements de textes ordinaires connus) qui met la forme des textes sous forme de titres et sous-titres. Il est disponible sous licence GNU GPL. La version utilisée est 0.7.1. C'est un projet relativement grand puisqu'il contient 65490 lignes de code et 198 classes et interfaces. La dernière version est 0.9, apparue en février 2011.

7.1.5 Lucene

Lucene est un moteur de recherche textuel. Il permet d'indexer et de rechercher de l'information dans du texte. C'est un projet logiciel libre soutenue par la fondation Apache mis à disposition sous licence [1]. Lucene est utilisé par plusieurs projets libres, des applications Webs, des produits tels Apple, IBM, plateforme Eclipse, AOL et Comcat. Nous utilisons la version 1.4 qui contient 197 classes et interfaces.

7.2 Identification des fonctionnalités issues de l'héritage multiple et de la délégation

Dans cette section, nous présentons les résultats de l'étude expérimentale de notre approche d'identification des fonctionnalités issues de l'héritage multiple et de la délégation à travers un agrégat. Pour chaque candidat, l'algorithme indique s'il est de type héritage multiple ou délégation. Une classe qui hérite d'une super-classe et implémente une interface, est considérée candidate pour l'identification d'une fonctionnalité issue de l'héritage multiple. De même, pour une classe qui implémente deux interfaces. Dans le cas d'une classe qui partage l'une ou plusieurs signatures de méthodes que l'un de ses attributs, elle est aussi candidate d'avoir une fonctionnalité issue d'une délégation à travers un agrégat.

L'analyse des résultats, regroupant les candidats pour l'héritage multiple et la délégation, a été effectuée manuellement, cas par cas afin de valider si un candidat représente effectivement une réutilisation de fonctionnalités par héritage multiple ou par délégation.

7.2.1 Héritage multiple

Nous appliquons l'algorithme décrit dans la section 4.1 sur les cinq outils. Après un filtrage des interfaces marqueurs[1] et des interfaces constantes, nous obtenons le tableau 7.II qui regroupe le nombre des classes ayant effectivement les fonctionnalités

[1]Une interface marqueur est une interface ne possédant pas de méthodes. Elle est utilisée pour marquer les classes qui l'implémente.

issues de l'héritage multiple dans chacun des logiciels étudiés. Des exemples illustrant le cas de l'héritage multiple sont présentés ci-dessous.

Outils	JHotDraw	JavaWebMail	Lucene	JReversePro	FreeMind
Version	5.2	0.7	1.4	1.4.1	0.7.1
Cas identifiés	29	23	10	24	13
Cas légitimes	22	23	4	10	13

Tableau 7.II – Les candidats de "l'héritage multiple"

- JHotDraw : pour JHotDraw, il existe plusieurs cas de classes qui héritent d'une classe et implémentent une ou plusieurs interfaces. Un bon nombre d'interfaces implémentées correspond aux interfaces *Listener* (voir figure 7.1). Par exemple, la classe `CompositeFigure` hérite de la classe `AbstractFigure` et implémente l'interface `FigureChangeListener`. C'est un patron commun dans le framework graphique basé sur le modèle d'événements Java. L'interface `Figure-ChangeListener` inclut plusieurs types de contrats que les figures graphiques ont besoin pour accomplir et pour réagir convenablement à un ensemble d'événements. Trois autres cas légitimes correspondent aux interfaces qui héritent de deux ou plusieurs interfaces, comme l'interface `ConnectionFigure` qui hérite des interfaces `Figure` et `FigureChangeListener`. Un autre exemple inclut la classe `BouncingDrawing` qui hérite de `StandardDrawing` et implémente l'interface `Animatable`;

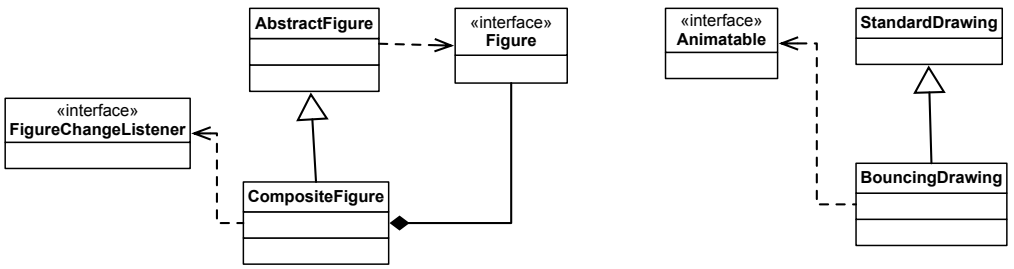


Figure 7.1 – Exemples de fonctionnalités supportées par héritage multiple dans JHotDraw

- JavaWebMail : dans le cas de JavaWebMail, les vingt-trois (23) candidats sont considérés comme appropriés. Une vingtaine (20) de ces cas correspond aux greffons (*plug-ins*) de JavaWebMail. Les greffons mettent en œuvre deux interfaces, `Plugin` et `URLHandler`. Les trois cas qui restent représentent une classe A qui hérite d'une classe B et implémente une interface telle la classe `WebMailServlet` hérite de `WebMailServer` et implémente l'interface `Servlet` ;

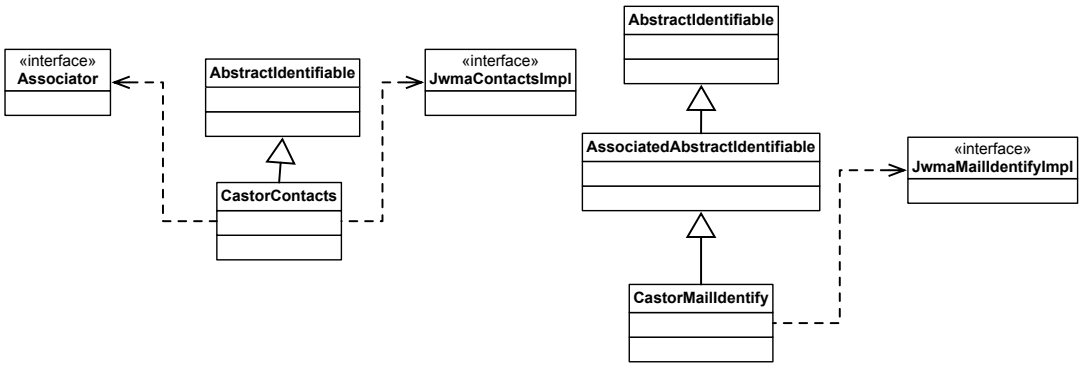


Figure 7.2 – Exemples de fonctionnalités supportées par héritage multiple dans JavaWebMail

- JreversePro : pour JreversePro, l'héritage multiple est identifié dans dix cas, tous ces cas valides sont issus de l'interface graphique GUI de JreversePro. Par exemple, le cas de la classe `JDlgFont`, qui hérite de la classe Dialog et implémente les interfaces `ItemListener` et `ActionListener` ;

- Freemind : treize (13) candidats pour l'héritage multiple sont considérés légitimes. Dix d'entre eux sont issus de l'interface GUI telle la classe `FreeMindApplet` qui hérite de `JApplet` et implémente l'interface `FreeMindMain`. Aussi, la classe `EdgeAdapter` qui hérite de `LineAdapter` et implémente `MindMapEdge`. Les trois cas qui restent correspondent à l'application du patron adaptateur comme la classe `LinkRegistryAdapter$ID_UsedStateAdapter` qui hérite de `LinkRegistryAdapter.ID_BasicStateAdapter` et implémente `MindMapLinkRegistry.ID_UsedState` ;

- Lucene : Pour le cas de Lucene, quatres cas légitimes correspondent à une combinaison de deux ou plusieurs fonctionnalités, comme la classe `SegmentTermPositions` qui hérite `SegmentTermDocs` et implémente `TermPositions`. Cependant, comme indiqué dans l'algorithme, nous ignorons les cas (six candidats dans notre cas) qui correspondent aux interfaces marqueurs, comme la classe `Query` qui implémente les interfaces `Serializable` et `Cloneable` (quatre cas), et les interfaces constantes, comme la classe `StandardFilter` qui hérite de `TokenFilter` et implémente `StandardTokenizerConstants`.

Après cette analyse, nous avons constaté qu'il y'a plusieurs différences entre les cinq outils. Pour le cas de JHotDraw, JavaWebMail et Lucene, les fonctionnalités héritées sont propres au domaine de l'application en question. Pour le cas de JreversPro et FreeMind, les interfaces héritées sont issues de l'interface GUI, et donc ne sont pas spécifiques à Freemind et moins à JreversePro. Le domaine d'application traite des expressions, des tables de symboles, des associations constantes, des nœuds de syntaxes, etc.

7.2.2 Délégation avec agrégation

Pour la délégation, nos résultats sont obtenus à travers deux processus. Le premier processus est automatisé : pour chaque paire <classe, attribut>, nous calculons le pourcentage de l'"interface métier" implémenté par la classe elle-même. L'"interface métier" d'un type sera l'ensemble de toutes ses méthodes (localement définies et héritées), pour lesquelles nous ignorons :

- les constructeurs ;
- les méthodes accesseurs (*getter*) i.e. "<some type> get<some name>()", et manipulateurs (*setter*) "void set<some name>(<param type> <param name>)" ;
- Les méthodes héritées des classes de l'extérieur du domaine d'application de la classe. Par exemple, les classes qui héritent de java.lang.Object, qui définissent les méthodes *hash()* et *equals(Object obj)* et autres. Il est difficile de reconnaître les méthodes qui proviennent des ancêtres de la classe à celles qui proviennent

des classes extérieurs au domaine. Par exemple, nous pouvons dire que les classes appartenant aux paquets java.* ou javax.* sont extérieures au domaine. Cependant, certains paquets incluent certaines fonctionnalités. Par exemple pour JHotDraw, les paquets javax.swing.* et java.awt.* sont pertinents au domaine de l'application. Le filtrage est réalisé manuellement et nous considérons que toutes les méthodes issues des ancêtres extérieurs au domaine seront exclues ;

Nous calculons le ratio suivant pour une paire <classe, attribut> :

$$ratio = \frac{taille(interface\_metier(attribut) \cap interface\_metier(classe))}{taille(interface\_metier(attribut))}$$

Un ratio égale à 1 signifie que toutes les méthodes du domaine de l'attribut sont implémentées par la classe qui le contient.

Le processus suivant est manuel et consiste d'analyser les candidats potentiels pour la délégation. Pour chaque méthode f() définie dans une classe et son attribut, nous vérifions s'il existe une délégation entre la classe et son attribut à travers l'implémentation de f(), par exemple :

```
class A {
private AttributeType attribut ;
...
ReturnType f() {
if (attribut != null) {
return attribut.f() ;
} else {
...
}
}
}
```

Les résultats préliminaires sont présentés au-dessous pour chaque outil. Nous analysons seulement les paires <classe, attribut> dont le ratio excède 50%. Dans certains cas,

nous avons pu identifier des cas parfaits.

7.2.2.1 JHotDraw

Le tableau7.III montre les résultats pour l'outil JHotDraw.

# cand. tel que ratio=100%	8
# cand. tel que 50% $\leq$ ratio <100%	7
# cand. tel que ratio < 50%	26
Totale cand. tel que ratio $\geq$ 50%	15
Cas de délégation effective	12

Tableau 7.III – Candidats de délégation à travers un agrégat pour JHtoDraw 5.2

Les cas parfaits (l'"interface du domaine" de l'attribut est inclus dans l'"interface du domaine" de la classe) sont divisés en trois catégories :

- Cas où le type de l'attribut est une interface qui est déjà implémentée par la classe, directement ou à travers une classe ancêtre (5 cas) ;
- Instantiation du patron singleton (deux cas). Dans ce cas, la classe a un membre de données statique qui est une instance de la classe. Par exemple, les classes `IconKit` et `ClipBoard`;
- Cas nommés fortuits : comme par exemple le cas de la classe `StandardDrawing-View`, qui a un attribut de type `PointConstrainer`. L'interface `PointConstr-ainer` a trois méthodes : *int getStepX(), int getStepY() et void constrainPoint(Point p)*, et la classe `StandardDrawingView` supporte la méthode *protected void constrainPoint(Point p)*. Comme les deux premières méthodes sont considérées comme des accesseurs par le calcul de l'"interface du domaine"[2], ce cas est qualifié comme un cas parfait.

[2]Nous choisissons d'exclure les méthodes getter/setter des interfaces du domaine pour ne pas encombrer de telles interfaces, lesquelles nous analysons les détails d'implémentation. Cependant, quand de telles méthodes apparaissent dans les *interfaces*, comme dans ce cas-ci, nous nous demandons si elles doivent être exclues de l'"interface du domaine". En fait, nous pouvons soutenir que l'"interface du domaine" d'une *interface Java* est l'interface elle-même

En inspectant les cas où une classe implémente le type d'un attribut, nous réalisons que "la délégation d'attribut" est utilisée dans différents cas :

- Une instantiation du patron décorateur. Ce qui est intéressant dans ce patron, la classe (le décorateur) qui représente le comportement modulaire qu'on souhaite ajouter à l'attribut (le type décoré) ;

- Un cas où la classe `SelectionTool`, qui implémente l'interface `Tool` à travers une classe ancêtre et délègue un comportement à l'attribut fChild de type `Tool`. La classe délègue à un attribut ce qui permet de changer de comportement durant son exécution, semblable au patron stratégie. Le comportement de `SelectionTool` dépend de ce qu'on sélectionne : l'arrière plan de la figure, une poignée sur la figure, ou la figure elle-même. L'attribut fChild permet de supporter plusieurs comportements, dépendamment des fonctionnalités sélectionnées. Dans ce cas, l'interface `Tool` représente une abstraction commune des différents comportements ;

- Un cas où la classe `TextFigure`, qui implémente l'interface Figure à travers ses ancêtres, délègue à l'attribut fObservedFigure de type `Figure`. L'attribut représente un modèle et la classe représente une vue, dans le contexte de MVC (Model View Controller) ;

- Deux cas où la classe `OffsetLocator` implémente l'interface `Locator` à travers ses ancêtres, ayant un attribut fBase de type `Locator`. Autre cas intéressant est celui de la classe `GraphicalCompositeFigure` comme dans la figure 7.3) semble être des instances du patron *chaîne de responsabilité*. Dans ces cas, les méthodes de la classe délèguent aux méthodes de l'attribut. L'auteur de la classe écrit dans le commentaire de la classe : "*The GraphicalCompositeFigure manages contained figures like the Composite Figure does, but delegates its graphical presentation to another (graphical) figure which ... mainly has a presentation purpose*". L'auteur n'aime donc pas le fait que la classe `CompositeFigure` combine les fonctionnalités d'un composite et ceux d'une figure dans le même ob-

jet. Il divise ces fonctionnalités qu'il aurait préféré voir séparées dans la classe `CompositeFigure`.

Aussi, nous avons analysé des cas imparfaits de délégation où le ratio varie de 75%, 66.66%, 62.5% et 50%. Certains cas sont décrits ci-dessous :

- L'interface `Storable` qui est héritée/implémentée par plusieurs interfaces/classes, "pollue" le résultat en gonflant le ratio de couverture car elle augmente la taille des "interfaces du domaine" d'application des attributs et des classes. Par exemple, la classe `PolyLineFigure`, qui implémente l'interface `Storable`, a deux attributs fStartDecoration et fEndDecoration de type l'interface `LineDecorator` qui hérite `Storable` et définit une seule méthode, *draw()*. Celle-ci est non implémentée par `PolyLineFigure`. Cependant, à cause de l'interface partagée `Storable` (avec ses méthodes *read()* et *write()*), nous avons obtenu un ratio de 66% ;

- Cas de méthodes obsolètes (*deprecated*) qui polluent les "interfaces du domaine". Ces méthodes peuvent augmenter les faux positifs (si la classe et l'attribut ont les mêmes méthodes obsolètes) et les faux négatifs (si seulement le type de l'attribut qui a ces méthodes) ;

- Deux cas qui auraient pu représenter des cas de délégation parfaite, se sont révélés ne pas être le cas car la méthode de la classe qui délègue et la méthode de l'attribut à laquelle se fait la délégation ont des signatures différentes. Nous étions conscients que cela peut arriver mais nous n'étions pas surs de son pourcentage ;

- Un "substantiel" comportement partagé entre les interfaces `Connector` et `Handle` révèle un cas qui a "échappé" aux concepteurs de JHotDraw. Les méthodes *draw(), displayBox(), containsPoint()* et *owner()* sont partagées par `Connector` qui a deux autres méthodes, et `Handle` qui a sept autres méthodes parmi lesquelles trois sont obsolètes.

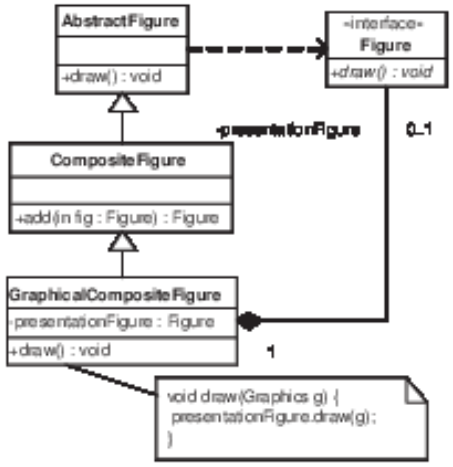


Figure 7.3 – Division de fonctionnalité d'une classe

Un autre cas (incluant l'implémentation du patron adaptateur ou *wrapper pattern*) est présenté par la classe abstraite `DecoratorFigure`, qui enveloppe la classe `Figure` qui lui délègue toutes les méthodes de `Figure`(figure 7.4).

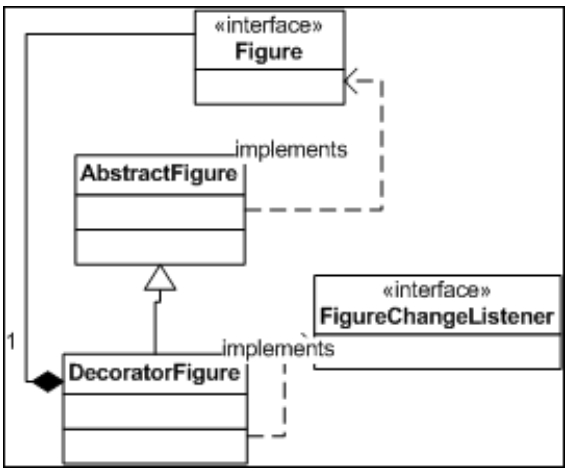


Figure 7.4 – Exemple de délégation dans JHotDraw

7.2.2.2 JavaWebMail

Le tableau 7.IV présente le résultat pour JavaWebMail. Un cas parfait qui correspond à l'instantiation du patron singleton : la classe `WebMailServer` avec l'instance du singleton `server`. Une autre instance pour la classe `FileStorage` et l'attribut logger

de type `Logger`, est un cas valide de délégation : les méthodes de `FileStorage` déléguées aux méthodes de l'attribut de type Logger. Le troisième cas, ayant un ratio de 66%, impliquant la classe `PluginDependencyTree` et l'attribut node de type `Plugin`, est une instance du patron composite, et peut être considéré comme un cas légitime de délégation[3].

# cand. tel que ratio=100%	2
# cand. tel que 50% $\leq$ ratio <100%	1
# cand. tel que ratio < 50%	13
Totale cand. tel que ratio $\geq$ 50%	3
Cas de délégation effective	2

Tableau 7.IV – Candidats de délégation à travers un agrégat pour JavaWebMail

7.2.2.3 JreversePro

Le tableau7.V présente le résultat pour JreversePro. L'algorithme identifie quatre cas. Comme mentionné auparavant, nous analysons seulement les candidats ayant la valeur du ratio supérieure ou égale à 50% (deux cas en tous). La classe `JInstruction` a deux attributs prev et next, pour la précédente et la prochaine instruction respectivement, sont aussi de type `JInstruction`. Dans ce cas, il n'y a pas de délégation.

# cand. tel que ratio=100%	2
# cand. tel que 50% $\leq$ ratio <100%	0
# cand. tel que ratio < 50%	2
Totale cand. tel que ratio $\geq$ 50%	2
Cas de délégation effective	0

Tableau 7.V – Candidats de délégation à travers un agrégat pour JreversePro

Aussi, nous avons décelé trois fonctionnalités dans cet outil - la récupération d'associations (constant pool retreival), le désassemblage et la décompilation- pour être déléguées aux différents composants d'un même objet. JreversePro utilise le patron visiteur

[3] `PluginDependencyTree` est une hiérarchie de plug-ins, où les fils du plugin *p* exige les services fournis par *p*. Dans ce cas, `PluginDependencyTree` délègue les méthodes `register(WebMailServer s)` et `String provides()`, qui concatènent les services fournis par tous les plugins de la hiérarchie

pour implémenter ces fonctionnalités. Les trois fonctionnalités sont déléguées à trois objets visiteurs différents, mais ces objets ne sont pas statiquement attachés à l'agrégat à travers les références des membres de données (attributs), ils sont des arguments de la méthode *acceptVisitor()* (figure 7.5).

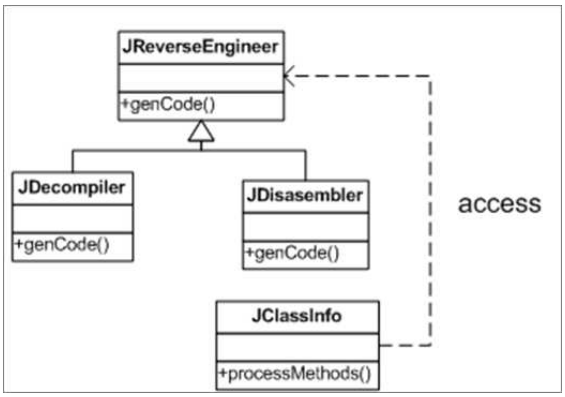


Figure 7.5 – Patron visiteur dans JReversePro

7.2.2.4 FreeMind

Le tableau7.VI présente le résultat pour l'outil Freemind. L'algorithme identifie vingt trois cas. Trois cas ayant comme ratio supérieur à 50%. Un cas parfait tel la classe `StylePattern`, qui a un attribut childStylePattern de type `StylePattern`. Cependant, après inspection, aucun comportement n'a été délégué à l'attribut. Deux autres candidats correspondent à la classe `NodeAdapter`, avec les attributs preferredChild et parent, les deux de types `MindMapNode`, qui est l'interface implémentée par `NodeAdapter`. Dans ce cas, les attributs sont utilisés comme une structure de liaison "pointeurs", sans délégation de comportement.

# cand. tel que ratio=100%	3
# cand. tel que 50% $\leq$ ratio <100%	0
# cand. tel que ratio < 50%	20
Totale cand. tel que ratio $\geq$ 50%	3
Cas de délégation effective	0

Tableau 7.VI – Candidats de délégation à travers un agrégat pour FreeMind

7.2.2.5 Lucene

Le tableau7.VII présente le résultat pour Lucene. L'algorithme identifie soixante-dix huit cas, incluant trente deux cas qui sont parfaits. En analysant les trente cinq cas légitimes, nous obtenons :

- Seize cas de délégation, incluant onze instances du patron adaptateur et une du patron proxy ;
- Dix neuf cas de faux positifs présentés dans les sous-catégories suivantes :
  - Onze cas de faux positifs correspondant aux cas où une classe de type T ayant une ou plusieurs attributs de type T aussi, utilisés pour créer les structures de données (listes). Dans tels cas, les attributs sont utilisés pour parcourir les listes, sans cas de délégation ;
  - Six cas de faux positifs correspondants aux attributs publiques statiques ;
  - Deux cas d'agrégation sans délégation.

# cand. tel que ratio=100%	32
# cand. tel que 50% $\leq$ ratio <100%	3
# cand. tel que ratio < 50%	43
Totale cand. tel que ratio $\geq$ 50%	35
Cas de délégation effective	16

Tableau 7.VII – Candidats de délégation à travers un agrégat pour Lucene

Un exemple de cas ayant un ensemble de méthodes déléguées est présenté par la figure 7.6. Par exemple, la majorité des méthodes de la classe `FilterIndexReader` délèguent aux méthodes de la classe `IndexReader`. Elles sont des classes d'accès aux indexes des documents.

7.2.2.6 Discussion

À partir de l'analyse ci-dessus, nous présentons certaines conclusions préliminaires :

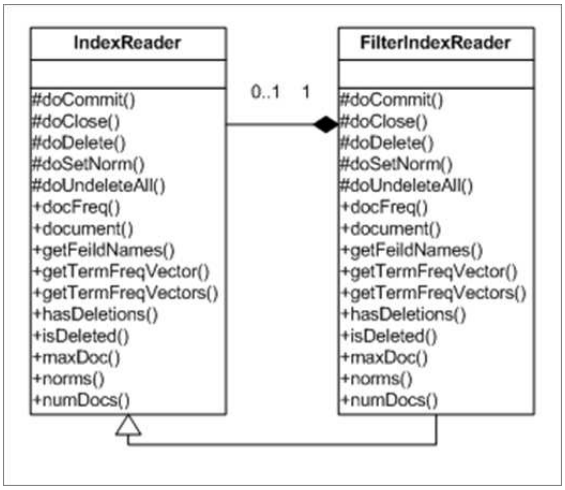


Figure 7.6 – Exemple de délégation dans le cas de Lucene

- La délégation à travers un agrégat a plusieurs utilisations qui ne se limitent pas seulement à celles illustrées par la figure 3.6 dans la section 3.3. Comme mentionné ci-dessus, elle est aussi utilisée dans plusieurs patrons de conception, incluant l'observateur, décorateur/adaptateur, le proxy, quelques variants du patron stratégie et chaîne de responsabilité. Bien que dans tous ces cas le type de l'attribut représente une fonctionnalité distincte et significative pour le domaine d'application, l'agrégation n'est pas nécessairement utilisée pour la composition de comportement.

- Notre calcul de l'interface du domaine et notre algorithme d'identification d'agrégation manquent de précision. Nous avons identifié un certain nombre de cas qui pouvaient être ciblés par "programmation", à savoir :

    1. Les instances de singleton, et plus généralement, les attributs statiques ;
    2. La prise en compte dans le calcul de certaines méthodes des interfaces de la catégories des accesseurs (getters et setters), mais qui représentent une valeur ajoutée dans leurs domaines d'application.
    3. Même si le repérage des méthodes obsolètes est facile, il n'est pas évident comment en tenir compte : devrait-elles être ignorées dans le calcul des inter-

faces du domaine des attributs/classes ? Que faire dans le cas où la délégation de traitement passe encore par ces méthodes obsolètes ?

4. Nous avons considéré comme schéma de délégation le cas où une classe partage des méthodes avec l'un de ses attributs. Cependant l'absence de ce schéma de délégation ne peut signifier l'absence de délégation. Une analyse basée sur les graphes d'appel serait appropriée pour avoir des résultats plus précis.

- Malgré les imprécisions (perfectibles) mentionnées ci-dessus, un ratio de couverture élevé n'est ni suffisant ni nécessaire pour conclure une composition de comportement via délégation. En effet, une classe A peut avoir besoin de seulement un sous-ensemble des fonctionnalités offertes par une classe B utilisée comme attribut. Le ratio de couverture entre A et B devrait refléter la proportion des fonctionnalités de B utilisée par A. Cependant, nous retenons que l'existence de fonctionnalités (méthodes) communes à une classe et l'un de ses attributs sans qu'il y ait pour toutes ces fonctionnalités une délégation de la classe vers l'attribut[4], pollue les résultats en donnant lieu à de faux positifs.

7.3 Identification des fonctionnalités issues de la multiplication d'états

À la recherche des fonctionnalités issues de la multiplication d'états, nous explorons les mêmes cinq outils utilisés auparavant. Nous avons commencé par implémenter notre algorithme présenté à la section 6.4.2. Les concepts candidats sont analysés manuellement cas par cas. Le tableau 7.VIII indique le nombre des concepts du treillis (colonne # conc.) généré à partir de la relation binaire décrite dans la section 6.4.1, le nombre total des concepts candidats (colonne # cand.) satisfont nos critères d'identification, ainsi que le nombre de candidats ayant *au moins* deux méthodes dans leurs intents (colonne #cand t.q #meth$\geq$2.). Le tableau 7.VIII qualifie aussi le résultat à travers une vérification manuelle des concepts candidats identifiés qui les sépare en 4 catégo-

[4]Comme c'est le cas de `Storable` dans JHotDraw ; Cas où une classe et son attribut hérite d'une même classe par exemple sans présence de délégation

ries (présentées en détails ci-dessous). La première catégorie couvre les fonctionnalités héritées d'une interface/super-classe (I/C), et celles acquises par délégation (D). Les deux sont représentées conjointement par la super-catégorie des candidats des fonctionnalités reconnues (# fonc.reconn.). Les candidats des fonctionnalités non reconnues (# fonc.nonreconn.Ad-hoc) sont partagés entre la catégorie ayant la structure similaire (S.S) et celle des structures non similaires d'implémentation (S.NS). Comme notre algorithme parcourt un treillis, Il identifie tous les concepts candidats à représenter une fonctionnalité. En effet, tout concept qui représente une fonctionnalité ou sous-fonctionnalité de celle-ci est comptabilisé parmi les candidats.

Outils	#conc.	#cand.	#cand t.q #meth≥2	#fonc. reconn.	#fonc. non reconn. Ad-hoc	
				I/C ou D	S.S	S.NS
FreeMind	1117	128	97	67	28	2
JavaWebMail	187	67	46	35	10	1
JHotDraw	595	375	336	318	17	1
JReversePro	370	25	16	8	6	2
Lucene	233	152	127	107	16	4

Tableau 7.VIII – Données expérimentales sur les différents logiciels.

7.3.1 Implémentation par héritage

L'implémentation par héritage couvre les candidats qui représentent une fonctionnalité issue d'une implémentation d'interface ou par l'héritage d'une super-classe. Ces candidats représentent généralement les fonctionnalités déjà reconnues par le développeur du système et qui ont été représentées par une interface ou une super-classe. Pour pouvoir réutiliser ces fonctionnalités, le développeur force la classe à implémenter l'interface concernée ou à hériter de la classe qui possède cette fonctionnalité. Cependant, notre approche présente une importante limitation. L'algorithme peut nous induire en erreur en ignorant les classes des librairies externes. En effet, dans certains cas, il peut signaler des classes candidats qui représentent des occurrences "d'une fonctionnalité". Après vérification manuelle, l'ensemble des membres sous-jacents a été en réalité hé-

rité d'une super-classe externe commune ou implémente une interface externe qui a été ignorée. Comme déjà cité, nous ignorons les méthodes issues de la librairie Java dans l'analyse du code.

Par exemple, dans JHotDraw, les méthodes *getPreferredSize(), getMinimumSize()* apparaissent dans les `ToolButton`, `StandardDrawingView`, et `Filler` (voir figure 7.7) qui n'ont pas une super-classe commune dans la hiérarchie de classes de l'application. Cependant, si on suit le lien de spécialisation dans JHotDraw, elles atteignent la classe `JComponent` de Java Swing. D'une manière remarquable, de tels cas sont facilement éliminés après le filtrage manuel des concepts candidats ayant des classes ayant des ancêtres communs.

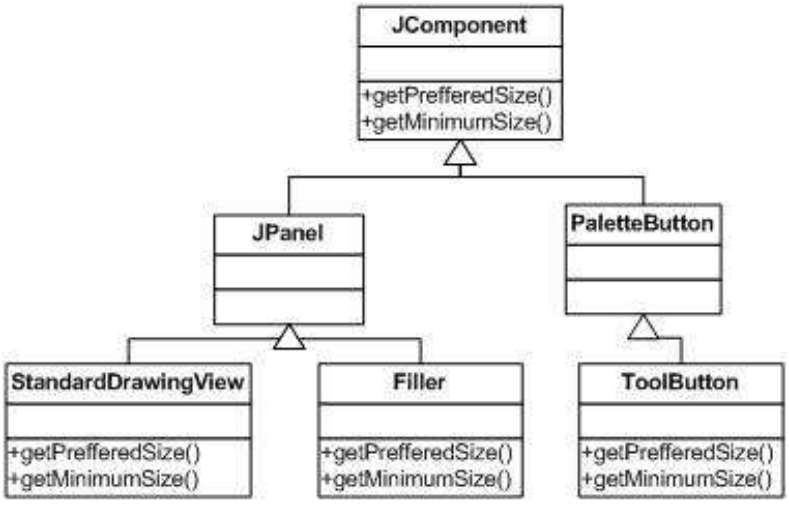


Figure 7.7 – Exemple d'héritage dans JHotDraw.

Aussi, comme notre algorithme a pour objectif d'identifier les patrons de codage utilisés pour concevoir et regrouper les fonctionnalités, il a pu identifié des cas de fonctionnalités regroupées sous forme d'interfaces. Par exemple dans JHotDraw, l'interface `DrawingView` est implémentée par les classes `StandardDrawingView` et `NullDrawingView`. Ainsi, *à fortiori*, `StandardDrawingview` et `NullDrawing-View` vont implémenter un ensemble substantiel de méthodes communes (voir figure 7.8).

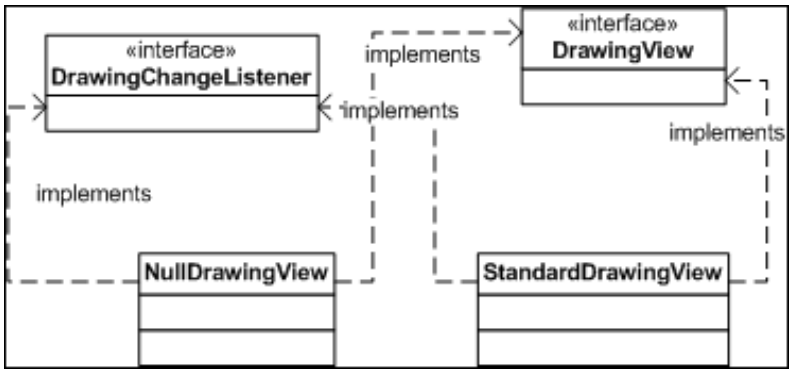


Figure 7.8 – Autre exemple d'implémentation basée sur héritage multiple dans JHotDraw : implémentation de plusieurs interfaces.

7.3.2 Implémentation par délégation

Dans le cas des fonctionnalités implémentées par délégation, notre algorithme identifie les candidats qui correspondent à des fonctionnalités qui ont été délibérés à travers un agrégat. Cependant, un cas nous a échappé, à l'algorithme aussi, est illustré dans la figure 7.3. Ce cas se présente quand notre algorithme identifie un candidat issu de l'héritage (si A hérite de B, A et B ne peuvent être comptés et considérés comme une occurrence indépendante), ou de la relation *implements* (si A et B implémentent l'interface C, A et B ne peuvent être considérés comme une occurrence indépendante). Il manque des subtilités à leur combinaison, les relations *implements* sont elles-mêmes héritées : si A hérite de B et B implémente C, alors A implémente aussi C. A cause de cette omission, l'algorithme nous a présenté par exemple les méthodes de l'interface `Figure` comme ayant des occurrences indépendantes dans `Figure` et `GraphicalCompositeFigure` (figure 7.3).

Certes, le cas de la figure 7.3 peut être facilement corrigé. Cependant, nous nous attendons à ce que certains programmes moins matures que JHotDraw puissent avoir des implémentations imparfaites de délégation où le composant et le composite n'implémentent pas explicitement, directement ou indirectement, la même interface comme dans la figure 7.3. Un exemple illustrant ce cas, la classe `Controller` et `MapView` de FreeMind. La classe `MapView` a un agrégat de type `Controller`. Les deux classes ayant les mêmes signatures de méthodes, mais sans délégation.

7.3.3 Implémentation ad-hoc

Notre approche vise principalement ce cas d'implémentation. L'algorithme a identifié un nombre de situations où deux classes, ou plus, partagent un sous-ensemble significatif de leurs APIs sans aucune relation entre elles, directement, comme l'héritage ou indirectement, via une interface commune implémentée. C'est le cas où un développeur, qui a probablement reconnu une similarité comportementale, a été assez discipliné pour choisir les mêmes signatures de méthodes dans deux places, mais il n'a pas formalisé cette similarité sous forme d'une interface commune ou d'un ancêtre commun.

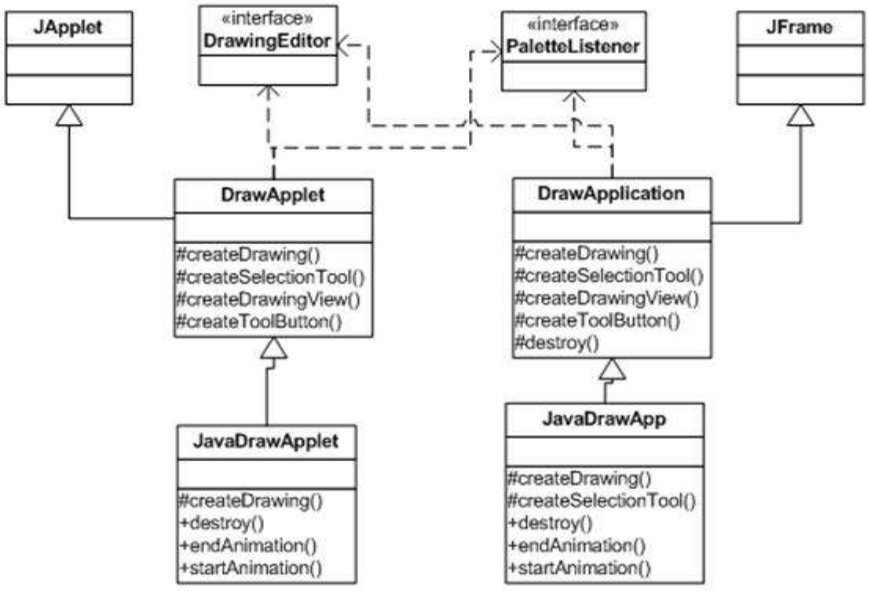


Figure 7.9 – Cas d'une implémentation ad-hoc de plusieurs fonctionnalités (JHotDraw)

La figure 7.9 montre un exemple de cela. Dans JHotDraw, la classe `DrawApplet` exécute l'application dans un navigateur (une Applet), `DrawApplication` dans une fenêtre (Frame). Les méthodes dans les classes `DrawApplet` et `DrawApplication` sont exactement *identiques* et pourtant elles n'appartiennent à aucune super-classe ou interface commune. De même pour les sous-classes `JavaDrawApplet` et `JavaDrawApp`. Dans ce cas, les développeurs peuvent regrouper les méthodes partagées dans une super-classe commune de `DrawApplet` et `DrawApplication`, qui peut être nommée

`DrawCreation` dans le but d'avoir une meilleure factorisation.

Un exemple dans FreeMind illustre ce type de cas (voir figure 7.10). La classe `ControllerAdapter` qui implémente l'interface `ModeController` et la classe `MapAdapter` qui implémente `MindMap`. `ControllerAdapter` et `MapAdapter` ont le même ensemble de méthodes {*load(), save(), getFrame(), getNext()*} sans être reconnu à travers une interface ou une super-classe commune.

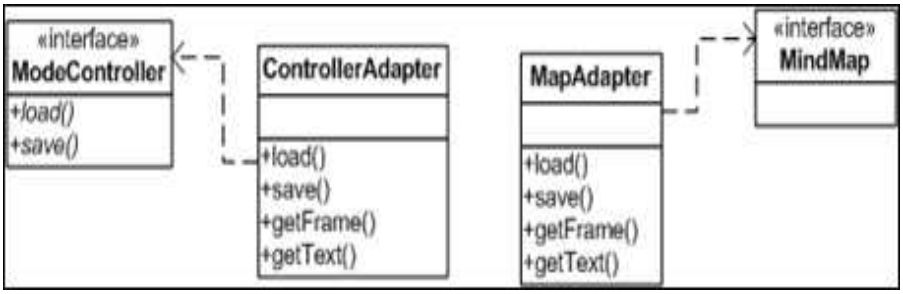


Figure 7.10 – Cas d'une implémentation ad-hoc dans FreeMind

Un autre exemple est montré dans la figure 7.11. Il montre *ce que signifie un gestionnaire de symboles*, appliqué à l'analyseur syntaxique de Lucene ; en d'autre termes, analyser le texte d'entrée (la saisie). C'est un cas extrême où les deux classes ont exactement la même API mais n'ont aucune relation qui les relie (héritage, implémentation d'interface ou délégation) !.

7.3.4 Exemple de fonctionnalités implémentées en mode structure non similaire

Nous avons défini une fonctionnalité par un ensemble de membres (méthodes) distribués entre différentes classes. Le positionnement hiérarchique des divers attributs et méthodes contribuent à une caractéristique donnée, comme celle illustrée par l'exemple de la figure 6.8 dans le chapitre précédent. Un autre exemple, (figure 7.12), montre le cas de multiplication d'états. Ici, `GraphicalCompositeFigure` est une figure composée de plusieurs autres figures qu'elle gère. `BouncingDrawing` est un conteneur pour toutes les figures dans le `Drawing`. La classe `AnimationDecorator` décore les figures avec une animation. `BorderDecorator` utilise les frontières comme déco-

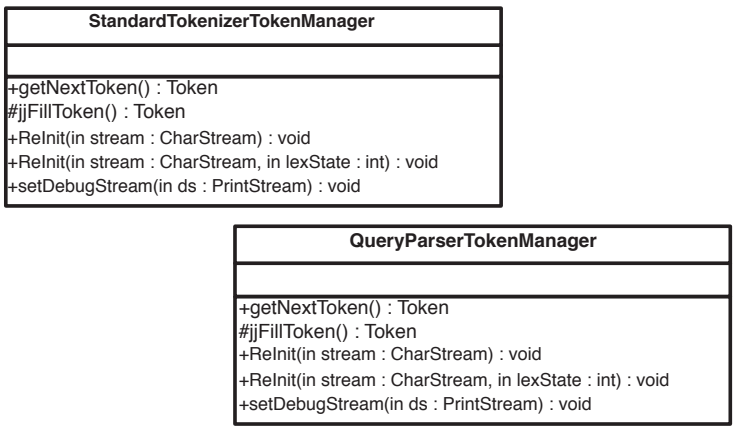


Figure 7.11 – Autre cas d'implémentation ad-hoc dans Lucene 1.4.

ration des figures. La méthode *animationStep()* actionne une étape d'animation, alors que la méthode *initialize()* lance l'initialisation. Il y a deux fonctionnalités combinées dans les deux sous-hiérarchies représentées par {`GraphicalCompositeFigure`, `BouncingDrawing`} et {`AnimationDecorator`, `BorderDecorator`}.

Un cas identifié par notre algorithme et qui illustre un contre exemple : les méthodes *drawBackground(), remove(), add() et addAll()* apparaissent dans le concept constitué de l'interface `DrawingView` et de la classe `AbstractFigure`. Toutes ces méthodes constitue l'interface `DrawingView`, alors qu'on les retrouvent éparpillées dans la sous-hiérarchie de `AbstractFigure` (voir figure 7.13).

Aussi, la figure 7.14 illustre un cas identifié par notre algorithme. Dans ce cas, l'algorithme a identifié un ensemble de méthodes {*cut(), edit(), setCloud(),...*} avec la classe `ControllerAdapter` et les sous-classes de `FreeMindAction`. Clairement, les classes n'ont pas la même API. Cependant, à cause des méthodes implémentées par les sous-classes, les agrégats interceptent tous les événements d'entrée, lesquels sont alors délégués aux composants spécialisés.

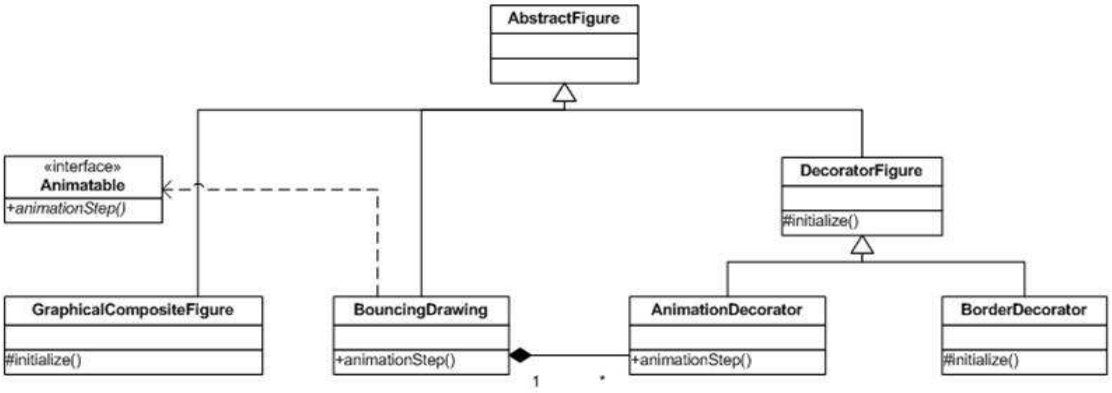


Figure 7.12 – Un cas d'implémentation de fonctionnalité sous forme de multiplication d'état dans JHotDraw.

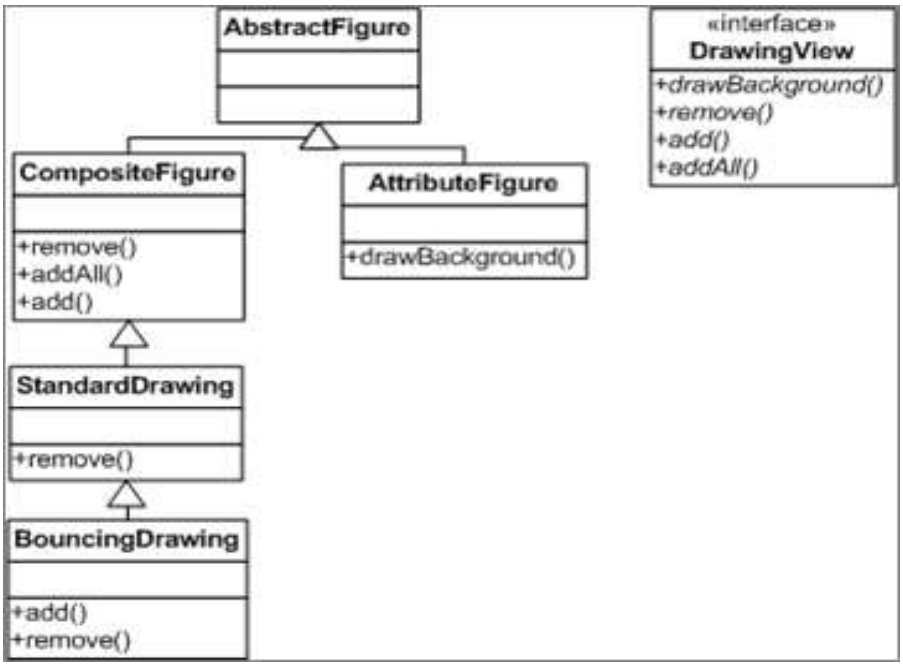


Figure 7.13 – Un cas d'implémentation de fonctionnalité avec structure non similaire dans JHotDraw.

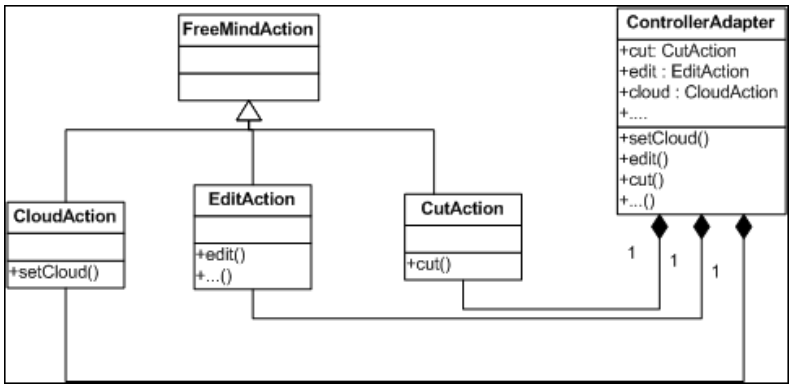


Figure 7.14 – Cas de délégation dans FreeMind

7.4 Analyse des résultats

La première approche d'identification des fonctionnalités issues de l'héritage multiple et de la délégation, nous a permis d'identifier explicitement les différentes fonctionnalités réutilisées. Elle nous aide explicitement à analyser et à comprendre le code.

Dans le cas où le développeur n'a pas pensé à appliquer ces techniques de réutilisation, notre deuxième approche nous a été de grand secours. Elle est pertinente dans le cas où le programmeur duplique le code partout où il en a besoin. C'est ce que nous avons appelé par "patrons récurrents" ou multiplication d'états.

Dans la deuxième approche, notre relation binaire nous a permis principalement une interprétation plus aisée des concepts du treillis (voir figure 6.26). Plusieurs cas sont repérables d'une manière claire et visible. Les cas de fonctionnalités déjà reconnues, par exemple les classes qui implémentent une même interface, sont facilement identifiés puisque le concept est constitué de l'interface et de ses classes associées dans son extent. L'extent est donc de la forme $\{I, C_1, .., C_n\}$. La vérification manuelle est toujours nécessaire, puisqu'il peut exister une classe dans cet extent qui n'implémente pas l'interface I. Dans ce cas, nous pouvons proposer que la classe implémente cette interface. Il en va de même pour le cas où il existe des méthodes communes entre deux hiérarchies de classes dont les racines implémentent deux interfaces indépendantes, par exemple de la forme $\{I_1, I_2, C_1, .., C_p\}$. Dans ce cas, nous avons des méthodes communes entre les deux interfaces et, bien évidemment, entre les classes qui les implémentent. Une suggestion

de factorisation entre ces interfaces est désirée aussi. Un autre point saillant détecté par notre méthode est le cas où il existe des méthodes communes entre interfaces sans que celles-ci ne soient implémentées par une classe. Ici aussi, une suggestion de factorisation entre ces interfaces est souhaitée ainsi de se demander sur l'utilité d'avoir des interfaces non implémentées par des classes. Bien évidemment, sans oublier l'identification des cas par héritage et par délégation. Par conséquent, la deuxième approche nous permet l'identification des fonctionnalités déjà reconnues par le concepteur, en plus des cas ad-hoc qui reflètent la mauvaise conception du code.

Un autre point important, notre deuxième approche peut nous aider à détecter les cas de délégation identifiés par la première approche. Nous pouvons avoir un concept dont l'intent contient un ensemble de méthodes et l'extent contient un ensemble de classes où il existe une classe qui délègue une partie de/toutes ces méthodes (existantes dans l'intent) à une classe ou plusieurs classes (de l'extent) qui est/sont comme un/des attribut(s) de la première classe.

7.5 Validité

Nous discutons les trois types de validité tels que définis dans [34] :

Validité interne : les menaces dans ce cas de validité concernent les facteurs qui peuvent affecter l'indépendance des résultats par rapport au chercheur sans que celui-ci s'en rende compte. Dans notre approche, l'étape de vérification manuelle où nous identifions des candidats à être des fonctionnalités (reconnues ou non reconnues) peut dépendre de la compétence et de l'interprétation du chercheur. Pour remédier à cette menace, nous recommandons l'utilisation d'un système de vote (par exemple dans un cadre de crowdsourcing) pour rendre l'identification des fonctionnalités à l'issue de la vérification manuelle insensible au chercheur.

Validité externe : nous nous sommes restreints à expérimenter nos approches sur des logiciels libres Java à défaut de la disponibilité des logiciels fermés. Les logiciels libres peuvent être développés par une équipe hétérogène ou même par une seule personne. Ceci peut être une menace à la validité externe vue que nous ne pouvons pas garantir la

généralisation des résultats à des cas de logiciels fermés où il peut y avoir des conventions de développement propre à une entreprise.

Validité de construction : elle concerne la généralisation de notre théorie à tous les logiciels légataires OO puisque nos approches se basent sur les techniques de réutilisation des fonctionnalités. Les résultats de notre étude restent préliminaires, bien qu'ils soient tout de même raisonnables. Nos approches ne sont pas complètes puisqu'il peut exister des fonctionnalités non extraites. De plus, certains résultats peuvent être des faux positifs, ce qui induit un manque de précision. Puisque nous ne disposons pas de cadre expérimental énumérant les fonctionnalités existantes dans les différents codes, nous n'avons pas pu mesurer la précision et la complétude de nos approches. Notre but était de savoir ce que l'on peut obtenir rien qu'avec les signatures.

7.6 Conclusion

D'après les expérimentations réalisées, nous déduisons que les fonctionnalités sont généralement partagées en deux catégories : celles qui ont été déjà reconnues par le concepteur comme celles implémentées sous forme d'interfaces, de super-classes ou en mode délégation, et celles non reconnues qui reflètent des cas de mauvaise factorisation ou de mauvaise conception. L'identification de ces dernières a été un ajout significatif et majeur pour notre travail. Par conséquent, la manière à travers laquelle nous avons défini notre approche pour identifier ces fonctionnalités, nous a montré une manière plus facile d'analyser et de comprendre la conception de code. Aussi, elle a rendu la tâche de vérification manuelle plus pratique. Une grande partie de notre travail a été basée sur l'identification des cas des fonctionnalités récurrentes, jugeant que la récurrence d'une manière ad-hoc (récurrence d'un ensemble de méthodes dans différents endroits sans être factorisé par une super-classe ou une interface) est une manière inadéquate pour réutiliser ces fonctionnalités existantes dans le code.

Nous avons appliqué nos approches sur un ensemble de logiciels libres différents de point de vue taille, type et degré de qualité. Même avec un niveau élevé de qualité de conception reconnue, par exemple dans le cas de JHotDraw, nous avons pu déceler des

éléments qui présentent une factorisation insuffisante.

CHAPITRE 8

CONCLUSION

8.1 Résumé

À fur et à mesure qu'un système logiciel évolue, le coût de sa maintenance augmente. La maintenance pose plusieurs défis. Il faut d'abord comprendre le système au niveau de sa conception logicielle. Il faut, ensuite, pouvoir identifier et localiser les parties du logiciel qui doivent être modifiées. Pour rendre plus facile cette activité, et réduire le coût de la maintenance, il est important de comprendre et de localiser les fonctionnalités du code. La localisation des fonctionnalités consiste à identifier les éléments (classes, méthodes, attributs) du code source qui implémentent une fonctionnalité donnée [105]. Cette identification constitue une tâche importante durant l'évolution du système [105]. Le but des techniques et des outils de localisation est de réduire l'espace de recherche que le développeur doit explorer. Dans notre travail, nous avons défini une fonctionnalité par un ensemble cohésif d'exigences fonctionnelles [117] [144]. L'objectif de notre recherche est d'identifier les éléments du code qui implémentent une fonctionnalité distincte dans un code légataire orienté objets *ne mettant pas en œuvre des techniques orientées aspects*. La question qui se pose alors est comment reconnaître ces éléments en l'absence des techniques orientées aspects.

Notre approche passe par l'identification des techniques orientées objet classiques qu'un développeur aurait pu utiliser pour intégrer différentes fonctionnalités dans du code orienté objets. Plus précisément, notre démarche consiste en trois étapes :

- identifier les techniques utilisées par les développeurs pour intégrer une fonctionnalité en l'absence des techniques orientées aspects ;
- caractériser l'empreinte de ces techniques sur le code ;
- et développer des outils pour identifier ces empreintes.

Ces techniques non-orientées aspects dépendent de plusieurs facteurs, tels la compétence du concepteur et son niveau d'expertise ainsi que la nature du langage de programmation utilisé (par exemple le langage supporte t-il l'héritage multiple ?). Ces techniques peuvent se manifester d'une manière assez mûre telle sa représentation à travers un patron de conception complexe élaboré tel le modèle MVC ou tout simplement à travers une interface au sens Java. Une autre manière moins développée et plus ad-hoc peut se manifester par des ensembles d'attributs et de méthodes qui apparaissent dans différents endroits d'une hiérarchie de classes. Généralement, les développeurs ont besoin d'effort additionnel pour comprendre comment les fonctionnalités sont implémentées et comment elles sont reliées entre elles.

Dans ce travail, nous avons identifié trois techniques non-orientées aspects pour intégrer des fonctionnalités dans du code légataire objet :

- l'héritage multiple, dans les langages qui le supportent– ou le simulent. Ainsi, une classe peut hériter des comportements différents de différents parents ;
- la délégation/agrégation, où une classe peut implanter plusieurs fonctionnalités en déléguant certaines de ces fonctionnalités à l'un de ses composants ;
- une technique par défaut, que nous avons appelée "ad-hoc", qui consiste à redéfinir le code de la fonctionnalité à chaque endroit où on en a besoin, un peu à la manière du «clonage de code».

Pour ce qui est de l'héritage multiple et de le délégation, ils laissent une empreinte assez "distincte" dans le code, qu'il est possible de détecter en se fiant uniquement aux signatures des classes : déclarations de classes pour l'héritage multiple, et déclarations de classes, d'attributs, et signatures des méthodes pour la délégation/agrégation. Comme nous l'avons expliqué dans le chapitre 4, malgré la relative simplicité de la démarche, un algorithme de détection doit prendre en compte différentes connaissances du domaine et des librairies du langage pour filtrer les faux positifs (*false positives*).

Pour ce qui est de l'implantation ad-hoc, il n'existe, malheureusement pas, d'empreinte structurelle distincte sur le code. Reconnaître qu'un ensemble de fonctions ou

attributs apparaissent à différents endroits d'une hiérarchie (ou forêt de hiérarchies) de classes peut-être posé en termes d'un problème de classification : on regroupe ensemble les parties de code qui font apparaître les mêmes ensembles d'attributs et de méthodes. C'est pour cela que nous avons pensé à utiliser l'analyse formelle de concepts (AFC). Elle permet des regroupements conceptuels de collection d'objets en se basant sur leurs propriétés communes. Ces regroupements sont les concepts formels composés d'ensembles d'objets partageant des spécifications. L'AFC a été reconnue comme particulièrement appropriée pour l'analyse et la refactorisation d'applications orientées objets [68, 116]. Plusieurs travaux de recherche en génie logiciel se sont basés sur l'AFC pour la remodularisation du code [15, 88, 132], l'identification des objets dans le code procédural [14] et la restructuration des hiérarchies de classes [66, 116, 135, 136].

Dans ce travail, nous avons envisagé une façon non standard pour définir la relation binaire pour la génération des treillis de Galois afin d'identifier les fonctionnalités dans le cas d'implantation ad-hoc [50, 51]. Cette relation est basée sur l'identification d'un ensemble récurrent de méthodes dans le code. Cette récurrence reflète les multiples utilisations de la fonctionnalité. Ainsi, notre relation binaire associe à chaque classe, l'ensemble des méthodes définies dans la classe et dans ses *sous-classes*. Cette relation a été la base pour la génération de la table de contexte du treillis de concepts ainsi que l'algorithme d'identification des fonctionnalités dans les applications légataires orientées objets. Elle nous permet une diminution du nombre de concepts dans le treillis à analyser et une interprétation plus facile des concepts candidats. Elle est détaillée dans le chapitre 6.

Pour valider nos algorithmes, nous les avons testés sur cinq applications logicielles à code source libre, de natures et de tailles différentes, touchant à différents domaines (infrastructure, logiciels, systèmes), et différents niveaux de maturité allant de JHotdraw, très bien conçu, puisqu'il est développé comme une "étude de cas dans les patrons de conception" à JavaWebMail, qui est jeune et "mal conçu". Les résultats étaient prometteurs pour les deux approches. Pour la première, nous avons décelé la plupart des fonctionnalités issues de l'héritage multiple et de la délégation à travers un agrégat (voir chapitre 7).

Nos expériences ont montré que l'algorithme basé sur l'AFC était bel et bien adapté à l'identification des cas ad-hoc. En plus de faciliter la compréhension du code, l'approche nous a permis de détecter des candidats à la refactorisation. Par exemple dans le cas de l'outil JHotDraw, dont la qualité de conception est reconnue, nous avons pu déceler des anomalies de conception qui méritaient une refactorisation.

8.2 Contributions

Pour une application légataire orientée objets, il est intéressant d'identifier et d'isoler des fragments de code qui implémentent une exigence ou tout simplement une fonctionnalité particulière. De telles identifications et localisations permettent une facilité de compréhension et de maintenance de l'application en cas de changement. Dans certains cas, il est avantageux de regrouper ces fonctionnalités en utilisant différentes techniques telles le *refactoring* [58] ou les techniques orientées aspects [111] afin de pouvoir les composer ou les réutiliser dans d'autres programmes. Notre travail s'inscrit dans ce cadre, et vise l'identification des fonctionnalités dans du code légataire orienté objets.

Différentes techniques de localisation de fonctionnalités ont été proposées : statiques, dynamiques et hybrides. Il existe deux types d'analyses statiques. L'analyse *textuelle* est une approche pour localiser les fonctionnalités. Elle détermine les similarités textuelles entre une demande formulée et des éléments du code source (méthodes) en utilisant des techniques de dépistage de l'information, dont le LSI [10, 17, 40, 41, 97, 113]. L'analyse *structurelle* fournit l'information sur différents types de dépendances dans le système. Elle se concentre sur l'invocation de méthodes dans un graphe de dépendance statique (PDG) [32, 75, 118, 119, 124, 147]. Habituellement, les techniques statiques structurelles exigent une connaissance minimale du système par les analystes afin de spécifier les parties du code source qui ont besoin d'être analysées.

Les techniques dynamiques, quant à elles, utilisent les traces d'exécution pour localiser les fonctionnalités dans des systèmes existants [105]. Sur le plan purement pratique, cette technique est souvent difficile à mettre en œuvre puisqu'il faut avoir accès à l'environnement réel d'exécution du code, ou encore, pouvoir le reproduire fidèlement en

laboratoire. Sur le plan algorithmique, la limitation majeure de ces techniques est due au fait qu'une seule exécution peut incorporer plusieurs fonctionnalités et il devient alors difficile d'isoler, dans une trace d'exécution, les appels reliés à une fonctionnalité particulière. Il existe aussi des techniques hybrides (statique et dynamique) [37, 47]. La représentation intermédiaire utilisée par les techniques dynamiques est regroupée par l'analyse de l'exécution des données. L'information statique est utilisée pour filtrer les traces d'exécutions. Ces approches ont leurs propres limitations [37, 47]

Notre approche se situe dans le contexte des approches statiques (structurelles). Cependant, parce que les empreintes que nous cherchions se limitaient aux signatures, notre approche est plus facile à mettre en œuvre que les autres approches structurelles qui dépendent d'une analyse poussée du code source (PDGs, techniques de slicing, etc.). Pour ce qui est du cas ad-hoc, le plus difficile à caractériser au niveau des interfaces, nous nous sommes appuyés sur l'AFC pour mettre en évidence les similarités dans le code. L'AFC a été beaucoup utilisée dans la recherche d'informations et dans l'extraction de connaissances. Elle permet de structurer les données sous une forme de regroupements hiérarchiques (treillis). Ayant une factorisation maximale, le treillis facilite la compréhension et la découverte de relations entre les données analysées. Aussi, une recherche guidée par un patron réduit l'espace de recherche.

Dans notre cas, nous avons envisagé une manière non standard pour définir la relation binaire pour la génération des treillis de Galois afin d'identifier les fonctionnalités [50, 51]. Notre contribution principale s'est manifestée par l'utilisation et l'application de cette relation binaire. Elle est originale car elle présente une nouvelle manière de représenter la hiérarchie de classes. Elle permet une diminution du nombre de concepts dans le treillis à analyser et une interprétation facile des concepts candidats. Elle est détaillée dans le chapitre 6.

Finalement, la validation de la technique d'identification pour le cas ad-hoc sur les outils choisis a révélé qu'elle englobe aussi la technique d'identification des cas de délégation puisqu'elle regroupe les classes ayant des signatures de méthodes communes indépendamment du type de la relation entre ces classes. La relation peut être une relation d'héritage, elle peut être une relation de délégation ou un cas ad-hoc.

8.3 Extensions et perspectives

La plupart des logiciels actuels répondent à diverses exigences, et donc implantant plusieurs fonctionnalités qui se chevauchent. L'objectif de notre thèse était de pouvoir isoler des parties du code qui implantent une fonctionnalité particulière. Celles-ci décrivent et définissent les services et les tâches que le système doit accomplir.

En termes d'extensions et perspectives à court terme, il est possible de raffiner nos deux approches : la première approche qui se base sur les techniques d'implanter les fonctionnalités par héritage multiple ou par délégation et la deuxième approche qui identifie les fonctionnalités récurrentes dans le code. Concernant la technique de délégation dans la première approche, il est possible d'introduire l'identification à travers non seulement l'agrégat mais aussi à travers les paramètres de la méthode. Un autre point à explorer dans ce cas-ci, est de ne pas se limiter aux signatures identiques des méthodes. Nous pourrions aussi considérer les signatures des méthodes similaires [104]. Concernant la deuxième approche, puisqu'elle identifie les cas ad-hoc et les cas de délégation en plus ceux de l'héritage, on peut envisager qu'elle puisse englober aussi le cas de l'héritage multiple puisque l'AFC permet de factoriser les spécifications communes. L'avantage de cette nouvelle approche serait l'homogénéité : un seul treillis nous indiquera tous les cas problématique/intéressants dans le code.

Aussi, nous envisageons à court terme que l'outil puisse analyser avec un plus fort degré d'automatisation et d'une manière plus avancée les cas des fonctionnalités candidats pour reconnaître les candidats issus de l'héritage simple d'une super-classe (ou multiple), de l'implémentation d'interface ou de la délégation. Elle nous permettra une amélioration au niveau de la localisation et au niveau de la présentation des résultats trouvés. Naturellement, le développeur peut uniquement vérifier la conformité des candidats identifiés. Concernant les candidats non reconnus par les types de patrons déjà cités, notre outil pourra indiquer si ces candidats sont *ad-hoc* ou ils sont présentés sous forme d'une structure hiérarchique. Cette extension aiderait largement le développeur à se retrouver dans le code et à gagner du temps au niveau de la fouille et de la compréhension du code. De plus, elle aiderait à mieux redocumenter le code au cas où la

documentation n'est pas présente.

Nous envisageons aussi à étudier le cas où la fonctionnalité soit plus complexe. Celle-ci se présente sous forme de deux parties : une partie de la fonctionnalité soit locale et une partie soit déléguée. Nous pensons utiliser l'analyse relationnelle de concepts [76].

Il est aussi envisageable d'améliorer l'interface graphique pour une visibilité supérieure et une meilleure clarté qui rendent l'identification plus facile et plus aisée. Ainsi, l'amélioration de la capacité d'aide à la navigation dans le treillis peut guider facilement l'utilisateur lors de l'exploration et l'analyse des résultats.

A moyen terme, nous envisageons que l'outil puisse proposer des modèles de factorisation dans le cas de la deuxième approche. Par exemple dans le cas où un candidat qui représente un élément récurrent dans le code (*ad-hoc*) (les classes ayant des signatures de méthodes communes), nous pourrons suggérer au développeur de regrouper ces méthodes dans une super-classe ou une interface. Le jugement du développeur est nécessaire dans ce cas selon la sémantique du code.

A long terme, nous projetons d'appliquer l'outil sur différentes versions d'un même logiciel pour voir son amélioration de point de vue sa qualité. Nous faisons l'hypothèse que plus le nombre des candidats *ad-hoc* diminue et le nombre des fonctionnalités reconnues augmente, meilleure est la factorisation du logiciel. Un tel critère indiquera que les fonctionnalités ont été réutilisées adéquatement à travers des structures de conception orientées objets connues et appropriées telles l'héritage, la délégation ou l'implémentation à travers une interface.

Par la suite, nous prévoyons étudier l'interaction entre les fonctionnalités identifiées. Cette interaction peut nous indiquer la dépendance entre les fonctionnalités existantes (les appels entre les méthodes des différentes fonctionnalités) et permettra d'avoir une idée sur le comportement du système. Dans le cas d'un changement ou d'une mise à jour, le développeur pourra donc facilement tenir compte de l'impact de ce changement sur les fonctionnalités participantes.

BIBLIOGRAPHIE

[1] URL http://en.wikipedia.org/wiki/Lucene.

[2] Aspectj. URL http://fr.wikipedia.org/wiki/AspectJ.

[3] Multi-dimensional separation of concerns : Software engineering using hyperspaces. URL http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.

[4] 2005. URL http://aspectwerkz.codehaus.org/.

[5] Maintenance du logiciel, octobre 2010. URL http://fr.wikipedia.org/wiki/Maintenance_du_logiciel.

[6] Programmation orientée aspect, 2011. URL http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_aspect#P

[7] Programmation structurée, fevrier 2011. URL http://fr.wikipedia.org/wiki/Programmation_structur%C3%A9e.

[8] Survey results : Applications with jhotdraw, Last updated 10 july 2002. URL http://www.jhotdraw.org/survey/survey_results.html.

[9] Marcus. A et Maletic. J. I. Identification of high-level concept clones in source code. Dans *ASE '01 : Proceedings of the 16th IEEE international conference on Automated software engineering*, page 107. IEEE Computer Society, 2001.

[10] Marcus. A et Maletic. J. Recovering documentation-to-source-code traceability links using latent semantic indexing. Dans *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, 0-7695-1877-X, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Michail. A. Browsing and searching source code of applications written using a gui framework. Dans *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, pages 327–337. ACM, 2002.

[12] Muller. H. A., S. R. Tilley, Orgun. M. A., Corrie. B. D. et Madhavji. N. H. A reverse engineering environment based on spatial and visual software interconnection models. Dans *SDE 5 : Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 88–98, New York, NY, USA, 1992. ACM.

[13] Popovici. A, Gross.T et Alonso.G. Dynamic weaving for aspect-oriented programming. *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, 2002.

[14] Sahraoui.H A., Lounis.H, Melo.W et Mili.H. A concept formation based approach to object identification in procedural code. *Automated Software Engg.*, Vol.6(4): 387–410, 1999.

[15] Van Deursen. A et Kuipers.T. Identifying objects using cluster and concept analysis. Dans *ICSE '99 : Proceedings of the 21st international conference on Software engineering*, pages 246–255. ACM, 1999.

[16] Abran.A, Moore.J, Bourque.P, Dupuis.R et Tripp.L. *Guide to the Software Engineering Body of Knowledge (SWEBOK®)*. ISBN 0-7695-2330-7. http ://www.swebok.org. IEEE Computer Society, 2004.

[17] Lucia. A.De, Fasano.F, Oliveto.R et Tortora.G. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, Vol.16(4):13, 2007.

[18] Aksit.M, Bergmans. L.M.J et Vural.S. An object-oriented language-database integration model : The composition-filters approach. Dans *ECOOP 92 European Conference on Object-Oriented Programming, Utrecht, The Netherlands, 1992*, volume 615, pages 372–395. Springer, 1992. URL `http://doc.utwente.nl/19412/`.

[19] Antoniol.G, Canfora. G, Casazza. G, De Lucia. A et Merlo. E. Recovering tra-

ceability links between code and documentation. *IEEE Trans. Softw. Eng.*, Vol.28 (10):970–983, 2002.

[20] Antoniol.G et Gueheneuc. Y-G. Feature identification : An epidemiological metaphor. *IEEE Trans. Softw. Eng.*, Vol.32(9):627–641, 2006.

[21] Arévalo.G. High-level views in object-oriented software using formal concept analysis, 2005 . URL `www.iam.unibe.ch/~arevalo/ppt/arevalo-phd.ppt`.

[22] Arévalo.G. Understanding behavioral dependencies in class hierarchies using concept analysis. Dans *Proceedings of LMO 2003 (Langages et Modeles a Object)*, pages 47–59, 2003.

[23] Arévalo.G. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. Thèse de doctorat, University of Bern, January 2005.

[24] Buchli.F Arévalo.G et Nierstrasz.O. Detecting implicit collaboration patterns. Dans *Proceedings of WCRE 2004 (11th Working Conference on Reverse Engineering)*, pages 122 – 131. IEEE Computer Society Press, November 2004.

[25] Nierstrasz.O Arévalo.G, Ducasse.S. X-ray views : Understanding internals of classes. Dans *Proceedings of ASE 2003*, pages 267–270. Montreal, Canada, IEEE Computer Society, October 2003.

[26] Bergmans.L et Aksit. M. Composing crosscutting concerns using composition filters. *Commun. ACM*, Vol.44(10):51–57, 2001.

[27] Biggerstaff.T.J., Mitbander.B. G. et Webster.D.E. Program understanding and the concept assignment problem. *Commun. ACM*, Vol.37(5):72–82, 1994. ISSN 0001-0782.

[28] Biggerstaff.T.J., Mitbander.B.G. et Webster.D. The concept assignment problem in program understanding. Dans *ICSE '93 : Proceedings of the 15th international*

conference on Software Engineering, pages 482–498. IEEE Computer Society Press, 1993.

[29] Bojic.D, Eisenbarth.T, Koschke. R, Simon. D et Velasevic. D. Addendum to "locating features in source code'. *IEEE Trans. Softw. Eng.*, Vol.30(2):140, February 2004.

[30] Brant.J. Jhotdraw. Master's thesis, 1995.

[31] Burke.B et Brock.A, 2003. URL `http://onjava.com/pub/a/onjava/2003/05/28/a`

[32] Chen.K et Rajlich.V. Case study of feature location using dependence graph. Dans *IWPC '00 : Proceedings of the 8th International Workshop on Program Comprehension*, 0-7695-0656-9, pages 241 – 247, Washington, DC, USA, 2000. IEEE Computer Society.

[33] Chen.K et Rajlich.V. Ripples : Tool for change in legacy software. *IEEE International Conference on Software Maintenance*, Vol. 0:230 – 239, 2001.

[34] Martin Höst Magnus C. Ohlsson Bjöorn Regnell Claes Wohlin, Per Runeson et Anders Wesslén. *Experimentation in Software Engineering An Introduction*. Kluwer Academic Publishers, 2000.

[35] Clarke.S, Harrison.W.H, Ossher.H et Tarr.P. Subject-oriented design : Towards improved alignment of requirements, design, and code. Dans *OOPSLA*, pages 325–339, 1999.

[36] Poshyvanyk. D, Petrenko.M, Marcus. A, Xie.X et Liu.D. Source code exploration with google. Dans *ICSM '06 : Proceedings of the 22nd IEEE International Conference on Software Maintenance*, 0-7695-2354-4, pages 334–338. IEEE Computer Society, 2006.

[37] Poshyvanyk. D, Gueheneuc. Y-G, Marcus.A, Antoniol.G et Rajlich.V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, Vol. 33(6):420–432, June 2007.

[38] Dagenais.B et Mili.H. Slicing functional aspects out of legacy code, submitted, 10 p.

[39] Dargham.J. *Principes et implantation de vues dans les langages Orientés-objets*. Thèse de doctorat, Universite de Montreal, 2001.

[40] Deerwester.S, Dumais.S.T, Furnas.G.W, Landauer.T.K et Harshman.R. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, Vol.41:391–407, 1990.

[41] Dekhtyar.A. Advancing candidate link generation for requirements tracing : The study of methods. *IEEE Trans. Softw. Eng.*, Vol.32(1):4–19, 2006. Member-Hayes, Jane Huffman and Student Member-Sundaram, Senthil Karthikeyan.

[42] Deprez.J-C et Lakhotia.A. A formalism to automate mapping from program features to code. Dans *IWPC '00 : Proceedings of the 8th International Workshop on Program Comprehension*, pages 69 – 78, Washington, DC, USA, 2000. IEEE Computer Society.

[43] Eaddy.M, Aho. A. V., Antoniol.G et Guéhéneuc.Y-G. Cerberus : Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. Dans *ICPC '08 : Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society.

[44] Eaddy.M, Zimmermann.T, Sherwood. K. D., Garg.V, Murphy.G.C., Nagappan. N et Aho. A. V. Do crosscutting concerns cause defects ? *IEEE Trans. Softw. Eng.*, Vol.34(4):497–515, 2008.

[45] Edwards.D, Simmons.S et Wilde.N. An approach to feature location in distributed systems. Dans *the Journal of Systems and Software. Industrial Tools ... SERC-TR-275 21*, volume Vol.79, pages 57–68, 2006.

[46] Egyed.A, Binder.G et Grunbacher.P. Strada : A tool for scenario-based feature-to-code trace detection and analysis. Dans *ICSE COMPANION '07 : Companion*

to the proceedings of the 29th International Conference on Software Engineering, pages 41–42, Washington, DC, USA, 2007. IEEE Computer Society.

[47] Eisenbarth.T, Koschke.R et Simon.D. Locating features in source code. *IEEE Trans. Software Eng.*, Vol.29(3):210–224, 2003.

[48] Eisenberg.A.D et De Volder.K. Dynamic feature traces : Finding features in unfamiliar code. *IEEE International Conference on Software Maintenance*, Vol.0: 337–346, 2005.

[49] ElKharraz.A, Mili.H et Valtchev.P. Mining functional aspects from legacy code. Dans *ICTAI '08 : Proceedings of the 2008 20th IEEE International Conference on Tools with Artificial Intelligence*, pages 403–412, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3440-4.

[50] ElKharraz.A, Valtchev.P et Mili.H. Using concepts analysis for mining functional features from legacy code. Dans *ICTAI '09 : Proceedings of the 2009 21th IEEE International Conference on Tools with Artificial Intelligence*, pages 625–629. IEEE Comp. Soc., 2009.

[51] ElKharraz.A, Valtchev.P et Mili.H. Concept analysis as a framework for mining functional features from legacy code. Dans Léonard Kwuida et Baris Sertkaya, éditeurs, *ICFCA 2010*, volume Vol.5986 de *Lecture Notes in Artificial Intelligence*, pages 267 – 282. Springer, 2010.

[52] Elrad.T, Aksit.M, Kiczales.G, Lieberherr.K et Ossher.H. Discussing aspects of aspect-oriented programming aop : Frequently-asked questions. *Communications of the ACM*, Vol.44(10):33–38, 2001. URL `http://doc.utwente.nl/37222/`.

[53] Elrad.T, Filman.R.E et Bader.A. Aspect-oriented programming - introduction. *Commun. ACM*, Vol.44(10):29–32, 2001.

[54] Wong.W. Eric et Gokhale.S. Static and dynamic distance metrics for feature-based code analysis. *Journal System Software.*, Vol.74(3):283–295, 2005.

[55] Fischer.M, Pinzger.M et Gall. H. Analyzing and relating bug report data for feature tracking. Dans *WCRE '03 : Proceedings of the 10th Working Conference on Reverse Engineering*, pages 90–99. IEEE Computer Society, 2003.

[56] Fiutem.R, Tonella.P, Antoniol.G et Merlo.E. A cliche'-based environment to support architectural reverse engineering. *IEEE International Conference on Software Maintenance,*, Vol.0:319–328, 1996.

[57] Forgac.M et Kollar.J. Static and dynamic approaches to weaving. Dans *5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics*, pages 201–210, January 25-26 2007.

[58] Fowler.M. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[59] Funk.P, Lewien.A, Snelting.G, Braunschweig.T et Softwaretechnologie.A. Algorithms for concept lattice decomposition and their applications. Rapport technique, 1995.

[60] Antoniol. G et Gueheneuc. Y-G. Feature identification : A novel approach and a case study. Dans *ICSM '05 : Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 357–366. IEEE Computer Society, 2005.

[61] Snelting. G. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. Softw. Eng. Methodol.*, Vol.5(2):146–189, 1996.

[62] Wilde.N. G, Gust.J.A. et Strasburg.T.D. Locating user functionality in old code. Dans *Proceedings, Conference on Software Maintenance*, pages 200–205, Orlando, FL, USA, 1992.

[63] Gamma.E, Helm.R, Johnson.R et Vlissides.J. Design patterns : Elements of reusable object-oriented software. Dans *Addison Wesley,*, 1995.

[64] Johnson.R Gamma.E, Helm.R et Vlissides.J.M. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wisley, 1995.

[65] Ganter.B. et Wille.R. *Formal Concept Analysis, Mathematical Foundations*. Springer, Berlin, 1999.

[66] Godin.R et Mili.H. Building and maintaining analysis-level class hierarchies using galois lattices. Dans *Proc. of OOPSLA'93*, pages 394–410. ACM Press, 1993.

[67] Godin.R, Mineau.R, Missaoui.R et Mili.H. Méthodes de classification conceptuelle basées sur les treillis de galois et applications. Dans *Revue d'intelligence artificielle*, volume Vol.9, pages 105–137, 1995.

[68] Godin.R et Valtchev.P. Formal concept analysis-based normal forms for class hierarchy design in oo software development. Dans *FCA : Foundations and Applications*, chaptitre 16, pages 304–323. Springer, 2005.

[69] Gold.N. Hypothesis-based concept assignment to support software maintenance. Dans *ICSM '01 : Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 545–548, Washington, DC, USA, 2001. IEEE Computer Society.

[70] Gold.N, Harman.M, Li.Z et Mahdavi.K. Allowing overlapping boundaries in source code using a search based approach to concept binding. *IEEE International Conference on Software Maintenance*, Vol.0:310–319, 2006.

[71] Grant.S, Cordy. J. R. et Skillicorn. D. Automated concept location using independent component analysis. Dans *WCRE '08 : Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 138–142, Wasington, DC, USA, 2008. IEEE Comp. Soc.

[72] Greevy.O, Ducasse.S et Girba. T. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. Dans *ICSM '05 : Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 347–356, Washington, DC, USA, 2005. IEEE Computer Society.

[73] Greevy.O, Ducasse. S et Girba.T. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. Dans *ICSM '05 : Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 347–356. IEEE Computer Society, 2005.

[74] Harrison.W et Ossher.H. Subject-oriented programming (a critique of pure objects). *Special issue of Sigplan Notice - Proceedings of ACM OOPSLA'93*, Vol.28 (10):411–428, 1993.

[75] Hill.E, Pollock.L et Vijay-Shanker. K. Exploring the neighborhood with dora to expedite software maintenance. Dans *ASE '07 : Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 14–23, New York, NY, USA, 2007. ACM Press.

[76] Huchard.M, Rouane Hacene.M, Roume.C et Valtchev.P. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, Vol.49(1-4):39–76, 2007.

[77] Maletic.J. I. et Marcus.A. Supporting program comprehension using semantic and structural information. Dans *ICSE '01 : Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[78] Bohnet. J et Doellner.J. Analyzing dynamic call graphs enhanced with program state information for feature location and understanding. Dans *ICSE Companion '08 : Companion of the 30th international conference on Software engineering*, pages 915–916, New York, NY, USA, 2008. ACM Press.

[79] Shiling.J. J. et Sweeney.P. F. Three steps to views : extending the object-oriented paradigm. Dans *OOPSLA '89 : Conference proceedings on Object-oriented programming systems, languages and applications*, pages 353–361. ACM, 1989.

[80] Jacobson.I et Pan-wei.N. *Aspect-Oriented Software Development with Use Cases*. 0321268881. Addison Wesley Publishing Company, 2005.

[81] Kiczales.G, Hilsdale. E, Hugunin.J, Kersten.M, Palm.J et Griswold. W.G. An overview of aspectj. Dans *ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[82] Kiczales.G, Irwin.J, Lamping.J, Loingtier.J.M, Lopes.C.V, Maeda.C et Mendhekar.A. Aspect-oriented programming. Dans *Proc. of ECOOP'97*, pages 220–242. Springer, 1997.

[83] Koschke.R et Quante.J. On dynamic feature location. Dans ACM, éditeur, *ASE'05*, pages 86–95, California, USA, November 7-11 2005.

[84] Kothari.J, Denton. T, Shokoufandeh. A et Mancoridis.S. Reducing program comprehension effort in evolving software by recognizing feature implementation convergence. Dans *ICPC '07 : Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 17–26, Washington, DC, USA, 2007. IEEE Computer Society.

[85] Kothari.J, Denton. T, Mancoridis.S et Shokoufandeh.A. On computing the canonical features of software systems. Dans *WCRE '06 : Proceedings of the 13th Working Conference on Reverse Engineering*, pages 93–102, Washington, DC, USA, 2006. IEEE Computer Society.

[86] Krone.M et Snelting.G. On the inference of configuration structures from source code. Dans *16th International Conference on Software Engineering*, pages 49–57. IEEE Computer Society Press, 1994.

[87] Kuhn.A, Ducasse. S et Gírba.T. Semantic clustering : Identifying topics in source code. *Inf. Softw. Technol.*, Vol.49(3):230–243, 2007.

[88] Kuipers.T et Moonen.L. Types and concept analysis for legacy systems. Dans *IWPC '00 : Proceedings of the 8th International Workshop on Program Comprehension*, pages 221–230. IEEE Computer Society, 2000.

[89] Laganière.R. URL `www.site.uottawa.ca:4321/~laganier/seg3510/patterns/`

[90] Lemieux.F. Langages de programmation. URL `www.uqac.ca/~flemieux/PRO102/Slides/Ch12.ppt`.

[91] Liu.D, Marcus.A, Poshyvanyk. D et Rajlich.V. Feature location via information retrieval based filtering of a single scenario execution trace. Dans *ASE '07 : Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 978-1-59593-882-4, pages 234–243, New York, NY, USA, 2007. ACM.

[92] Liu.H. et Lethbridge. T. Intelligent search techniques for large software systems. Dans *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, pages 10–18. IBM Press, 2001.

[93] Aksit. M. Separation and composition of concerns in the object-oriented model. *ACM Comput. Surv.*, Vol.28(4):148, December 1996.

[94] Pigoski. T. M. *Practical Software Maintenance*. John Wiley & Sons, New York, 1996.

[95] Marcus.A et Rajlich.V. Identifications of concepts, features, and concerns in source code. Dans *IEEE International Conference on Software Maintenance*, volume Vol.0, page 718, 2005.

[96] Marcus.A, Rajlich.V, Buchta.J, Petrenko.M et Sergeyev. A. Static techniques for concept location in object oriented code. Dans *IWPC '05 : Proceedings of the 13th International Workshop on Program Comprehension*, pages 33–42, Washington, DC, USA, 2005. IEEE Computer Society.

[97] Marcus.A, Sergeyev.A, Rajlich.V et Maletic.Jonathan. An information retrieval approach to concept location in source code. Dans *WCRE '04 : Proceedings of the 11th Working Conference on Reverse Engineering*, 0-7695-2243-2, pages 214–223. IEEE Computer Society, 2004.

[98] Mcheick.H. *Distribution d'objets avec les techniques de développement orientées aspects*. Thèse de doctorat, Universite de Montreal, 2006.

[99] Meyer.B. *Conception et programmation par objets pour de logiciel de qualité*. Paris, 1990.

[100] Mili.H, Dargham.J et Mili.A. Views : A framework for feature-based development and distribution of oo applications. Dans *HICSS*, pages 8049–, 2000.

[101] Mili.H, Dargham.J, Cherkaoui.O Mili.A et Godin.R. View programming for the decentralized development of oo programs. Dans *Proceedings of TOOLS 99, Santa Barbara, CA., August,*, pages 210 – 221, 1999.

[102] Mili.H, Mcheick.H, Dargham.J et Sadou.S. Corbaviews : Distributing objects with views. *ACS/IEEE International Conference on Computer Systems and Applications*, pages 369–378, 2001.

[103] Mili.H, Sahraoui.H, Lounis.H, Mcheick.H et ElKharraz.A. Concerned about separation. Dans *Proc. of FASE'06*, pages 247–261. Springer, 2006.

[104] Navarro.G. A guided tour to approximate string matching. *ACM Comput. Surv.*, Vol.33(1):31–88, 2001.

[105] Norman.W et Scully. M.C. Software reconnaissance : Mapping program features to code. *Journal of Software Maintenance : Research and Practice*, Vol.7(1): 49–62, 1995.

[106] Olszak.A et Jørgensen. B. Remodularizing java programs for comprehension of features. Dans *FOSD '09 : Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 19–26, New York, NY, USA, 2009. ACM.

[107] Ossher.H, Kaplan.M, Harrison.W, Katz.A et Kruskal. V. Subject-oriented composition rules. *SIGPLAN Not.*, Vol.30(10):235–250, 1995. ISSN 0362-1340.

[108] Ossher.H et Tarr.P. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. *International Conference on Software Engineering,*, Vol.0:687–688, 1999. ISSN 0270-5257.

[109] Ossher.H et Tarr.P. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, Vol.44(10):43–50, 2001.

[110] Ossher.H et Harrison. W. Combination of inheritance hierarchies. Dans *OOPSLA '92 : conference proceedings on Object-oriented programming systems, languages, and applications*, pages 25–40, New York, NY, USA, 1992. ACM.

[111] Monteiro.M. P et Fernandes.J. M. Towards a catalog of aspect-oriented refactorings. Dans *Proc. of AOSD'05*, pages 111–122. ACM Press, 2005.

[112] Paul.S, Prakash.A, Buss. E et Henshaw.J. Theories and techniques of program understanding. Dans *CASCON '91 : Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 37–53. IBM Press, 1991.

[113] Poshyvanyk.D et Marcus. A. Combining formal concept analysis with information retrieval for concept location in source code. Dans *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 37–48, June 2007.

[114] Pressman.R.S. *Software Engineering, A Practitioner's Approach*. McGraw-Hill, 2001.

[115] Quintian.L. *JADAPT : Un Modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet*. Thèse de doctorat, Ecole doctorale « Science et Technologie de l'Information et de la Communication » de Nice Sophia-Antipolis, Thèse de Doctorat, 13 Juillet 2004.

[116] Godin. R., Mili. H., Mineau. G. W., Missaoui. R., Arfi. A. et Chau.T. T. Design of class hierarchies based on concept (galois) lattices. *Theory and Application of Object Systems*, Vol.4(2):117–134, 1998.

[117] Turner.C. R, Fuggetta. A, Lavazza. L et Wolf.A.L. A conceptual basis for feature engineering. *J. Syst. Softw.*, Vol.49(1):3–15, 1999.

[118] Rajlich.V. A methodology for incremental change. Dans *Extreme Programming Perspective, Marchesi,M., Succi, C., Wells, D., and Willaims, L., Eds.*, pages 201–214. Addison Wesley, 2002.

[119] Rajlich.V et Gosavi. P. Incremental change in object-oriented programming. *IEEE Softw.*, Vol.21(4):62–69, 2004.

[120] Revelle.M. et Poshyvanyk. D. An exploratory study on assessing feature location techniques. Dans *ICPC '09 : Proceedings of 17th IEEE International Conference on Program Comprehension*, pages 218 – 222, Vancouver, BC, Canada, 2009.

[121] Robillard.M, Shepherd.D et Hill.E.K. An empirical study of the concept assignment problems. Rapport technique, School of Computer Science, McGill University, June 2007.

[122] Robillard.M.P. *Representing concerns in source code*. Thèse de doctorat, The University of British Columbia (Canada), 2003.

[123] Robillard.M.P. Automatic generation of suggestions for program investigation. Dans *ESEC/FSE-13 : Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 11–20. ACM, 2005.

[124] Robillard.M.P. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, Vol.17(4):1–36, 2008.

[125] Robillard.M.P et Murphy.G. Feat : A tool for locating, describing, and analyzing concerns in source code. Dans *In Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, 2003.

[126] Rohatgi.A, Hamou-Lhadj.A et Rilling.J. An approach for mapping features to code based on static and dynamic analysis. Dans *ICPC '08 : Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 236–241, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3176-2.

[127] Letovsky. S. et Soloway. E. Delocalized plans and program comprehension. *IEEE Softw.*, Vol.3(3):41–49, 1986.

[128] Salah.M et Mancoridis.S. A hierarchy of dynamic software views : From object-interactions to feature-interactions. Dans *ICSM '04 : Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 72–81, 2004.

[129] Schach.S.R. *Object-Oriented and Classical Software Engineering*. 2002.

[130] Shao.P et Smith.R.K. Feature location by ir modules and call graph. Dans *ACM-SE 47 : Proceedings of the 47th Annual Southeast Regional Conference*, pages 1–4, New York, NY, USA, 2009. ACM.

[131] Shepherd.D, Fry.Z.P., Hill.E, Pollock.L et Vijay-Shanker.K. Using natural language program analysis to locate and understand action-oriented concerns. Dans *AOSD '07 : Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224, New York, NY, USA, 2007. ACM. ISBN 1-59593-615-7.

[132] Siff.M et RepsT.W. Identifying modules via concept analysis. Dans *ICSM '97 : Proceedings of the International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society, 1997.

[133] Simmons.S, Edwards.D, Wilde.N, Homan.J et Groble.M. Industrial tools for the feature location problem : an exploratory study : Practice articles. *J. of Software Maintenance : Research and Practice*, Vol.18(6):457–474, 2006.

[134] Snelting.G. Concept lattices in software analysis. Dans *Proc. Formal Concept Analysis*, pages 272–287, 2005.

[135] Snelting.G et Tip.F. Reengineering class hierarchies using concept analysis. Dans *Proceedings of ACM SIGPLAN/SIGSOFT Symposium on Foundations of Software Engineering*, pages 99–110, Orlando, FL, 1998.

[136] Snelting.G et Tip.F. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, Vol.22(3):540–582, 2000.

[137] Tarr.P, Ossher.H, Harrison.W et Sutton.S. N degrees of separation : multi-dimensional separation of concerns. Dans *Proc. of ICSE'99*, pages 107–119, 1999.

[138] Tonella.P et Ceccato.M. Aspect mining through the formal concept analysis of execution traces. Dans *WCRE '04 : Proceedings of the 11th Working Conference on Reverse Engineering*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society.

[139] Karakostas. V. Intelligent search and acquisition of business knowledge from programs. *Journal of Software Maintenance*, Vol.4(1):1–17, 1992.

[140] Vlissides.J.M et Schmidt.D.C, éditeurs. *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, Vancouver, BC, Canada, October 24-28, 2004. ACM. ISBN 1-58113-831-8.

[141] Wilde.N, Buckellew.M, Page.H et Rajlich.V. A case study of feature location in unstructured legacy fortran code. Dans *CSMR '01 : Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 68 – 76, Washington, DC, USA, 2001. IEEE Computer Society.

[142] Wilde.N, Buckellew. M, Page.H, Rajlich.V et Pounds.L. A comparaison of methods for locating features in legacy software. *J. Syst. Softw.*, Vol.65(2):105–114, 2003.

[143] Wille.R. Tensorial decomposition of concept lattices. *Mathematics and Statistics, Springer Netherlands ISSN*, Vol.2(1):81–95, March 1985.

[144] Wong.E.W, Gokhale.S.S, Horgan.J.R et Trivedi.K.S. Locating program features by using execution slices. Dans *ASSET '99 : Proceedings of the 1999 IEEE Sym-*

posium on Application - Specific Systems and Software Engineering and Technology, pages 194–203, Washington, DC, USA, 1999. IEEE Computer Society.

[145] Ye.Y et Fischer.G. Supporting reuse by delivering task-relevant and personalized information. Dans *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, pages 513–523, New York, NY, USA, 2002. ACM.

[146] Maarek .Y.S, Berry.D.M et Kaiser.G.E. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, Vol.17(8):800–813, 1991.

[147] Zhao.W, Zhang.L, Liu.Y, Sun.J et Yang.F. Sniafl : Towards a static non-interactive approach to feature location. *ACM Trans. Softwware Engineering and Methodologies*, Vol. 15(2):195–226, 2006.