Détection des préoccupations transversales par l'analyse formelle de concepts des diagrammes de séquence

Fairouz DAHI, Nora BOUNOUR

Laboratoire LISCO

Département d'Informatique

Université Badji-Mokhtar

BP. 12, 23000, Annaba

ALGÉRIE

fairouz_dahi@yahoo.fr, nora_bounour@yahoo.fr

RÉSUMÉ. L'existence des préoccupations transversales enchevêtrées ou dispersées, rend complexe la compréhension et l'évolution du code source orienté objet. L'adoption industrielle du paradigme orienté aspect a donné naissance à la recherche de nouvelles approches supportant la migration vers le paradigme orienté aspect. Cette migration a eu besoin d'extraire les préoccupations transversales afin de les encapsuler dans des aspects. Dans cet article, nous proposons, une nouvelle approche d'identification des préoccupations transversales au niveau conceptuel. Nous matérialisons ce dernier par les diagrammes UML de classes et de séquence. Nous utilisons l'analyse formelle des concepts, afin de regrouper les fonctionnalités dispersées au niveau des diagrammes de séquence, et nous analysons l'ordre d'appels des méthodes, afin de détecter celles enchevêtrées. Nous filtrons ensuite l'ensemble des aspects candidats obtenus, afin d'éviter les erreurs.

ABSTRACT. The existence of crosscutting concerns tangled or scattered, complicates the understanding and evolution of object oriented source code. The industrial adoption of aspect-oriented paradigm has led to research new approaches supporting aspect oriented migration. This migration requires the identification of crosscutting concerns, in order to encapsulate them into aspects .We propose in this paper a new approach for the identification of crosscutting concerns at the conceptual level. We materialize this latter by the UML class and sequence diagrams. We use the formal concept analysis to group scattered functionalities in sequence diagrams, and we analyze the order of method calls to detect the tangled ones. Then, we filter all obtained candidate aspects, in order to avoid the mistakes.

MOTS-CLÉS : Analyse d'ordre d'appels, Analyse formelle des concepts, Aspect, Extraction d'aspects, Préoccupations transversales, Réingénierie, Rétro-ingénierie.

KEYWORDS: Aspect, Aspect mining, Call order analysis, Crosscutting concerns, Formal concept analysis, Reengineering, Reverse engineering.

Received, March 31, 2013
Revised, December 31, 2013
Accepted, January 7, 2014

ARIMA Journal, vol. 18 (2014), pp. 19-35.

1. Introduction

Toutes les études en génie logiciel ont montré que la maintenance et l'évolution sont les phases les plus longues et les plus coûteuses dans le cycle de vie du logiciel [9]. Afin d'éviter la dégradation de l'efficacité du logiciel, il est nécessaire de satisfaire de nouvelles exigences des utilisateurs (commerciales, technologiques, etc.), ou modifier celles existantes.

A cet effet, la maintenance des logiciels a fait l'objet d'une grande attention ces dernières années. Elle est utilisée pour apporter des modifications sur le logiciel déjà implémenté. Mais auparavant, on doit faciliter le processus de compréhension des logiciels existants, afin d'en améliorer la maintenance.

Pour comprendre un logiciel complexe, on doit se baser sur une information digne de confiance. Celle-ci est représentée principalement par le code source du logiciel. La compréhension du code source d'un logiciel est facilitée par l'amélioration de sa structure. Une amélioration possible consiste en la migration des systèmes logiciels vers d'autres paradigmes jugés meilleurs du point de vue structuration, compréhension et réutilisation.

La programmation orientée objet (POO) encapsule les données et les traitements associés dans des classes. Ces dernières peuvent être corrélées. Malgré que la modularisation offerte par les classes ait pour but de rendre les classes indépendantes entre elles, on constate que des fonctionnalités enchevêtrées, telles que les contraintes d'intégrité référentielle, viennent se superposer à ce découpage et brisent l'indépendance des classes. La POO ne fournit aucune solution pour prendre en compte proprement ces fonctionnalités [4].

En POO, le mécanisme principal d'interaction entre objets est l'invocation de méthodes. Un objet qui souhaite effectuer un traitement invoque une méthode d'un autre objet. Un objet peut aussi invoquer une de ses propres méthodes. Dans tous les cas, il existe un rôle d'invocateur et un rôle d'invoqué. Par conséquent, en POO, l'implémentation d'une méthode est localisée dans une classe, tandis que son invocation, ou utilisation, est dispersée. Ce phénomène de dispersion du code est un frein à la maintenance et l'évolution des applications orientées objets. Toute modification dans la manière d'utiliser un service entraîne des modifications nombreuses, coûteuses et sujettes à l'erreur [4].

L'intérêt porté à la migration des systèmes orientés objet vers le paradigme orienté aspect se justifie, d'une part, par le choix, pour plusieurs projets logiciels, de la programmation orientée objet; d'autre part, par la complexité de la compréhension du code source objet, due à l'existence des préoccupations enchevêtrées ou dispersées (appelées préoccupations transversales) telles que la distribution et la synchronisation. Ces dernières représentent des extensions comportementales transversales à la hiérarchie des classes, et brisent leur indépendance. Le paradigme orienté aspect [19] sépare de telles préoccupations transversales (*crosscutting concerns*) dans de nouvelles unités de modularisation appelées *aspects.*

La dispersion (*Scattering*) se produit quand le code d'une même fonctionnalité est distribuée à travers le système (au niveau implémentation, un fragment de code similaire est distribué à travers de nombreux modules du programme), tandis que l'enchevêtrement (*Tangling*), c'est quand deux ou plusieurs fonctionnalités sont contenues dans une même autre fonctionnalité (au niveau implémentation, deux ou plusieurs préoccupations sont mises en œuvre dans le même corps du code d'un module, ce qui le rend plus difficile à comprendre) [18].

Afin de bénéficier des avantages du paradigme orienté aspect, il est nécessaire de développer des approches et des outils qui identifient et séparent les préoccupations

transversales (rétro-ingénierie). L'identification des préoccupations transversales au niveau implémentation souffre de certaines limites [1, 17]. Le code source est lié à une plateforme spécifique. En outre, sa compréhension demeure difficile, du fait de sa complexité et sa longueur. Ainsi, les traces d'exécution reflètent réellement le déroulement des exécutions des tâches du système. Cependant, les approches qui identifient les préoccupations transversales à partir des traces d'exécution sont incomplètes, car la trace d'un programme concerne un ensemble particulier des données d'un programme. Une analyse complète est presque impossible d'exploiter tous les chemins possibles d'exécution, ce qui rend les aspects candidats identifiés incomplètes.

Dans cet article, nous proposons une nouvelle approche d'identification des préoccupations transversales au niveau conceptuel. Ce dernier constitue un niveau plus abstrait, indépendant de la technologie d'implémentation. Nous le matérialisons par les diagrammes UML de classes et de séquence [10]. Ces derniers doivent être détaillés et finaux, afin qu'ils reflètent directement l'implémentation du logiciel.

Le diagramme de séquence documente les interactions entre objets pour achever un résultat. L'ordre chronologique des différentes interactions peut être une bonne source pour détecter l'enchevêtrement. Ainsi, le diagramme de séquence constitue une version proche de la version finale qui représente l'ensemble des fonctionnalités du système. En d'autres termes, les objets et les méthodes invoquées durant les transmissions des messages dans les diagrammes de séquence d'un système reflètent, respectivement, les objets et les méthodes de l'implémentation de ce système. Le diagramme de classes permet de cerner l'ensemble des méthodes et des classes qui existent dans le système, et sert à voir si le contexte d'invocations de méthodes (par transmissions des messages) est enchevêtré ou dispersé.

Notre approche est à trois passes. En une première passe, nous utilisons l'analyse formelle des concepts, qui est une méthode de regroupement conceptuelle, afin de regrouper les fonctionnalités dispersées. Dans une deuxième passe, nous analysons l'ordre d'appels des méthodes, afin de détecter les fonctionnalités enchevêtrées. Nous nous proposons ensuite dans une troisième passe de filtrer l'ensemble des aspects candidats obtenus (à partir de ces deux analyses), afin d'éviter les erreurs dans le processus de découverte d'aspects.

La section 2 présente les concepts de base du paradigme orienté aspect. Une nouvelle approche d'identification des préoccupations dispersées et enchevêtrées, ainsi que le filtrage de ces préoccupations, feront l'objet de la section 3. Une étude de cas sera présentée dans la section 4. La section 5 exploite quelques travaux connexes. Nous terminerons notre article par une conclusion qui nous permettra d'ouvrir des perspectives pour des développements futurs.

2. Les concepts orientés aspects

Le paradigme orienté aspect se base sur la séparation des préoccupations, tel que les objets métiers sont développés dans des classes, comme dans le paradigme orienté objet, mais indépendamment des préoccupations transversales. Ces dernières sont développées d'une façon complètement indépendante, et encapsulées dans des unités, appelées aspects. Un aspect est une unité modulaire qui implémente une préoccupation transversale [13]. Les concepts de base formant l'aspect sont les points de jonction et les consignes.

Un point de jonction est un point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés [4]. Un point de coupure désigne un ensemble de ponts de jonction [4]. La consigne est un bloc de code qui définit le comportement d'un

aspect. En outre, il est possible de spécifier à quel moment cette consigne doit être exécutée, avec l'un des types suivants :

- Le type *before* : une consigne de type *before* implique que son code est exécuté avant les points de jonction.
- Le type *after* : une consigne de ce type implique que son code est exécuté après les points de jonction.
- Le type *aroun*d : ici, le code est exécuté avant et après les points de jonction [4].

3. Vers une nouvelle approche de détection de la transversalité

L'UML est devenu une référence intéressante dans le domaine du génie logiciel. Sa richesse et sa puissance d'expression le rendent également puissant pour la modélisation [16]. Dans cet article, nous allons identifier les aspects en utilisant les modèles de conception matérialisés par les diagrammes UML de classes et de séquence.

Le diagramme de classes constitue un élément très important de la modélisation. Il permet de structurer le travail de développement, de manière très efficace, en identifiant la structure des classes d'un système, y compris les propriétés et les méthodes de chaque classe.

Le diagramme de séquence est un diagramme comportemental qui décrit les interactions entre objets. Il met l'accent sur l'ordre chronologique dans lequel s'effectuent les échanges de messages entre les objets. Dans la plupart des cas, la réception d'un message est suivie par l'exécution d'une méthode d'une classe (exceptés par exemple les messages de transmissions de données, messages d'erreurs, etc.).

A partir des diagrammes de séquence, nous allons utiliser l'ordre chronologique de transmissions de messages afin d'identifier les préoccupations enchevêtrées, tandis que les invocations répétitives des méthodes pendant les transmissions des messages sont utilisées afin d'identifier les préoccupations enchevêtrées.

Afin d'illustrer notre approche, nous prenons un exemple simplifié d'une bibliothèque en ligne, qui contient deux scénarios: emprunter et restituer. Le diagramme de classes est illustré par la figure 1. Les figures 2 et 3 présentent les diagrammes de séquences des cas d'utilisation emprunter et restituer, respectivement.

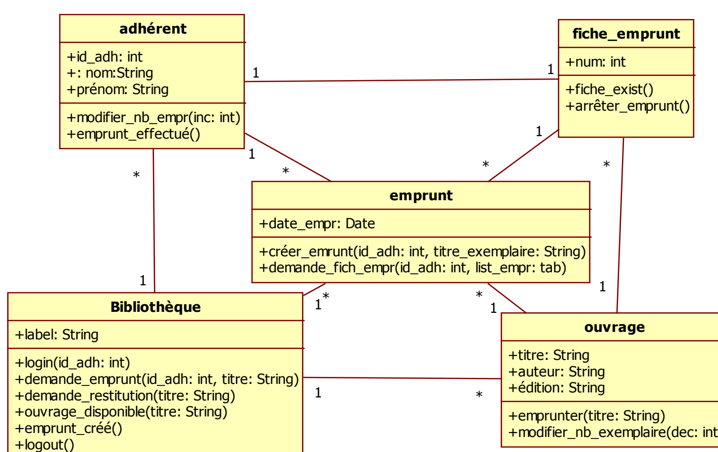

Figure 1. *Diagramme de classes du système de bibliothèque en ligne*

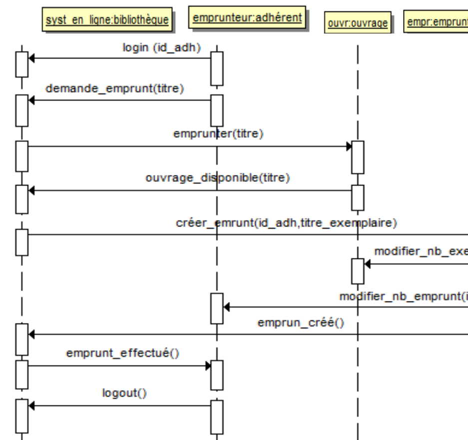

Figure 2. *Diagramme de séquence du cas d'utilisation « emprunter »*

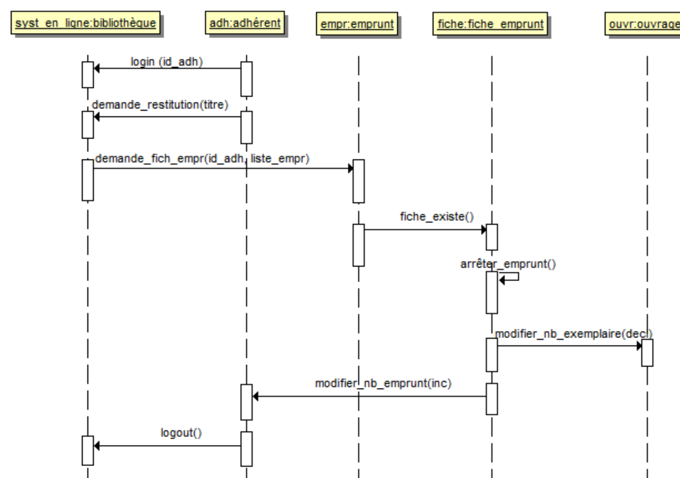

Figure 3. *Diagramme de séquence du cas d'utilisation « restituer »*

Selon les diagrammes de séquence, l'ensemble des objets existant dans le système de la bibliothèque en ligne est :

O1) `syst_en_ligne`
O2) `adh`
O3) `ouvr`
O4) `empr`
O5) `fiche`
O6) `emprunteur`

En se basant sur le diagramme de classes, l'ensemble des méthodes existant dans le système de la bibliothèque en ligne est :

m1) `emprunt_effectué()`
m2) `modifier_nb_empr(inc: int)`
m3) `login(id_adh: int)`
m4) `demande_emprunt(id_adh: int, titre: String)`
m5) `demande_restitution(titre: String)`
m6) `ouvrage_disponible(titre: String)`
m7) `emprunt_créé()`
m8) `logout()`
m9) `créer_emprunt(id_adh: int, titre_exemplaire: String)`
m10) `demande_fich_empr(id_adh: int, list_empr: tab)`
m11) `fiche_exist()`
m12) `arrêter_emprunt()`
m13) `emprunter(titre: String)`
m14) `modifier_nb_exemplaire(dec: int)`

La séquence (ordre) d'exécution des méthodes du système à partir des diagrammes de séquence est :

Cas d'utilisation : Emprunter

1. O6(m3)
2. O6(m4)
3. O1(m13)
4. O3(m6)
5. O1(m9)
6. O4(m14)
7. O4(m2)
8. O4(m7)
9. O1(m1)
10. O6(m8)

Cas d'utilisation : Restituer

1. O2(m3)
2. O2(m5)
3. O1(m10)
4. O4(m11)
5. O5(m12)
6. O5(m14)
7. O5(m2)
8. O2(m8)

3.1. Détection de la dispersion

Durant l'analyse des diagrammes de séquence, notre approche proposée met l'accent seulement sur les messages qui invoquent les méthodes. Les entités de base à analyser sont alors les méthodes et leurs invocations. Vu que ces entités resteront au niveau implémentation, les aspects détectés au niveau des diagrammes de séquence apparaissent dans l'implémentation comme attendus.

Les interactions (messages transmis) entre les différents objets du diagramme de séquence reflètent les invocations de méthodes du système. Ces méthodes sont décrites par le diagramme de classes UML de ce système. Un diagramme de séquence capture le

comportement d'un seul scénario de cas d'utilisation, pour cette raison, nous utiliserons l'ensemble des diagrammes de séquences du système logiciel.

Nous utilisons l'analyse formelle des concepts [6] afin de détecter les préoccupations dispersées, en exploitant les messages transmis entre les objets dans les diagrammes de séquence.

L'analyse formelle des concepts utilise les données d'une relation binaire, et exploite les propriétés de la correspondance, afin d'en isoler une hiérarchie de concepts, dits formels [6]. Les principaux éléments qui interviennent dans l'AFC sont :

- Le contexte de l'AFC : il est représenté par le triplet (O, A, R), où O désigne l'ensemble d'objets, A est l'ensemble des attributs, et R décrit la relation binaire qui relie les objets à leurs attributs.
- La matrice de l'AFC : le contexte de l'AFC est structuré dans une matrice binaire *M*, tel que : si un objet i possède la propriété j, $M[i, j] = 1$, sinon $M[i, j] = 0$.
- Le concept de l'AFC : un concept est la paire (X, Y), où X est un ensemble d'objets, et Y les attributs de ces objets. Un concept associe un ensemble maximal d'objets, à l'ensemble d'attributs que ces objets partagent.
- Le treillis de concepts : ou treillis de *Galois*, est une hiérarchie de concepts. Ces derniers sont ainsi organisés : un attribut, présent dans un concept, est hérité par tous les sous-concepts, et inversement pour les objets.

Nous utilisons l'AFC afin de détecter la dispersion au niveau des diagrammes de séquence:

- Le contexte de l'AFC : concernant le contexte (O, A, R), nous utilisons l'ensemble des objets O pour représenter les objets du système qui existent dans les diagrammes de séquence, et l'ensemble des attributs A pour représenter toutes les méthodes du diagramme de classes, invoquées par ces objets, c'est-à-dire les méthodes invoquées durant les transmissions des messages dans les diagrammes de séquence. La relation binaire R désigne la relation d'appels entre les objets et leurs méthodes invoquées, représentées par la transmission des messages dans les diagrammes de séquence.
- La matrice de l'AFC : concernant la matrice binaire *M*, les lignes contiennent les noms des objets (qui apparaissent dans les diagrammes de séquence), et les colonnes contiennent les noms des méthodes, tel que : si l'objet i transmet le message qui invoque la méthode j (l'objet *i* est l'appelant) alors $M\ [i, j] = 1$, sinon $M[i, j] = 0$.
- Le concept de l'AFC : dans un concept (X, Y), X est un ensemble des objets appelants, et Y est l'ensemble des méthodes appelées par ces objets.
- Le treillis de concepts : il regroupe les méthodes par appels. Il nous aide à déterminer les méthodes dont l'appel est dispersé.

Plus nous remontons dans le treillis construit à partir de la matrice *M*, plus nous trouvons des concepts contenant les méthodes dont leurs invocations sont dispersées. Ceci s'explique par le fait que les concepts qui se situent en haut du treillis contiennent plus d'objets, et donc leurs méthodes peuvent être des préoccupations dispersées si les objets sont des instances de différentes classes.

Les aspects candidats sont tous les attributs (méthodes appelées) de tous les concepts du treillis, qui ont le nombre de leurs objets (nombre d'objets appelants) supérieur à 1 (métrique *fan-in* > 1 [7]), car la refactorisation (en aspect) [12] d'une méthode appelée une seule fois n'améliore pas la structure du système. Pour la même raison, une méthode dont l'appel n'est pas dispersé, c'est-à-dire elle est appelée par le même objet ou appelée par

l'objet qui la contient, n'est pas un aspect. Nous proposons l'algorithme suivant (figure 1) pour la détection de la dispersion.

```
Soit {(Obj),(Meth)} un concept du treillis, et
Classes l'ensemble de classes différentes dont
Obj sont leurs instances.
Si |Classes|>1 Alors m (m ϵ Meth) est un
aspect candidat dispersé.
```

Figure 1. *Algorithme de détection de la dispersion à partir du treillis de concepts*

En appliquant l'AFC sur l'exemple de bibliothèque en ligne illustré ci dessus, la matrice de l'AFC obtenue est la suivante (Table 1).

Table 1. *Matrice M de l'AFC du système de la bibliothèque en ligne*

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12	m13	m14
O1	1	0	0	0	0	0	0	0	1	1	0	0	1	0
O2	0	0	1	0	1	0	0	1	0	0	0	0	0	0
O3	0	0	0	0	0	1	0	0	0	0	0	0	0	0
O4	0	1	0	0	0	0	1	0	0	0	1	0	0	1
O5	0	1	0	0	0	0	0	0	0	0	0	1	0	1
O6	0	0	1	1	0	0	0	1	0	0	0	0	0	0

Le treillis des concepts obtenu à partir de la matrice de l'AFC (table 1) est le suivant (figure 4) :

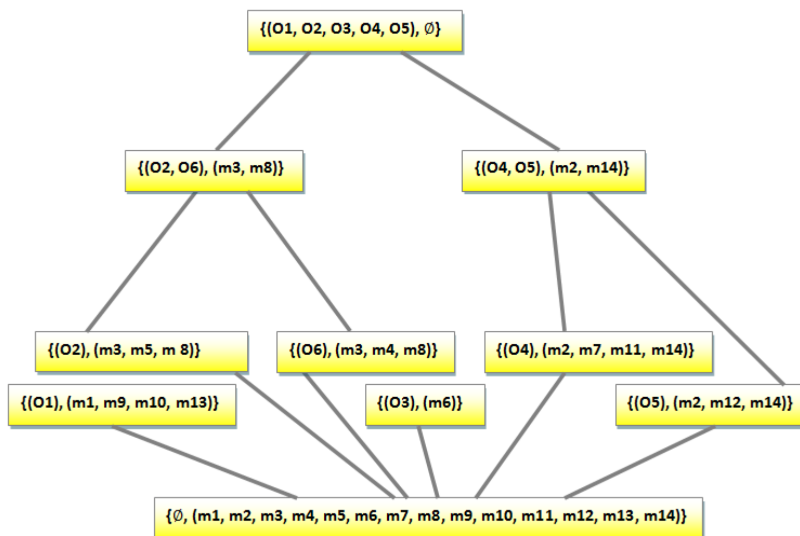

Figure 4. *Treillis de concepts résultant de la matrice « M »*

En interprétant le treillis de concept ci-dessus (figure 4), les méthodes ***m2 et m14*** (appelées par O4 et O5), et les méthodes ***m3 et m8*** (appelées par O2 et O6) sont des aspects

candidats, car elles sont appelées plus d'une fois par des objets de différentes classes. Les méthodes détectées par l'AFC sont des consignes d'aspects candidats.

3.2. Détection de l'enchevêtrement

Les interactions entre les objets dans les diagrammes de séquence reflètent un ordre chronologique d'exécution des tâches du système. L'analyse de l'ordre d'appels vise à utiliser cet ordre chronologique, afin de détecter les modèles récurrents d'appels de méthodes via les transmissions des messages. Par conséquent, il sera possible de détecter les préoccupations enchevêtrées.

Dans cet article, nous utilisons « type d'aspect » pour désigner le type de la consigne d'aspect. Les aspects trouvés par l'analyse de l'ordre d'appels seront de type *before* ou *after* :

- Si, dans les diagrammes de séquence, le message qui implique l'invocation de la méthode *a* est transmis toujours avant celui qui implique l'invocation de la méthode *b*, il y aura un aspect de type *before*, dont la méthode *a* est sa consigne, tandis que *b* représente son point de coupure.
- Si le message qui implique l'invocation de la méthode *a* est transmis toujours après celui qui implique l'invocation de la méthode *b*, alors il existe un aspect de type *after*, dont la méthode *a* représente sa consigne, et la méthode *b* représente son point de coupure.
- Si le message qui implique l'invocation de la méthode *a* est transmis toujours avant celui qui implique l'invocation de la méthode *b*, et ce dernier est transmis toujours après le message qui implique l'invocation de la méthode *a*, l'aspect *before* dont *a* représente la consigne et l'aspect *after* dont *b* représente la consigne sont symétriques. En d'autres termes, nous disons qu'il existe un aspect *before* (*a* est sa consigne et *b* son point de coupure) ou il existe un aspect *after* (*b* est sa consigne et *a* son point de coupure).

Notre analyse de l'ordre d'appels facilite la refactorisation des aspects, en précisant la consigne de l'aspect, son type et ses points de coupure.

Afin de concrétiser notre détection de l'enchevêtrement, nous utilisons une matrice carrée $(n * n)$ appelée « Martice d'Enchevêtrement ME », tel que n est le nombre de méthodes, et $ME[i, j]$ est le nombre de fois qu'une paire de méthodes successives (*i, j*) est invoquée par les transmissions des messages au niveau des diagrammes de séquence. Si $ME[i, j] > 1$ alors il existe un modèle récurrent.

Pour chaque cellule $ME[i, j] > 1$, soit $X = \sum_{k=1}^{n} | ME[i, k] \neq 0 , k \neq j |$ le nombre de cellules dans la ligne i ayant la valeur différente de zéro, et $Y = \sum_{k=1}^{n} | ME[k, j] \neq 0 , k \neq i |$ est le nombre de cellules dans la colonne j ayant la valeur différente de zéro. Nous proposons l'algorithme de la figure 5 pour identifier les aspects enchevêtrés à partir de la matrice ME. Si la méthode de la ligne i et celle de la colonne j appartiennent à deux classes différentes, alors nous appliquons l'algorithme de la figure 5.

```
Si X = Y = 0 alors {
          Type= symétrique;
Sinon si X = 0 alors {
          Type= after;
          Consigne= j;
          Point de coupure= i;
          }
Sinon if Y = 0 alors {
          Type= before;
          Consigne= i;
          Point de coupure= j;
          }
```

Figure 5. *Algorithme d'identification d'enchevêtrement à partir de la matrice ME*

Table 2. *Matrice ME du système de la bibliothèque en ligne*

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10	m11	m12	m13	m14
m1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
m2	0	0	0	0	0	0	1	1	0	0	0	0	0	0
m3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
m4	0	0	0	0	0	0	0	0	0	0	0	0	1	0
m5	0	0	0	0	0	0	0	0	0	1	0	0	0	0
m6	0	0	0	0	0	0	0	0	1	0	0	0	0	0
m7	1	0	0	0	0	0	0	0	0	0	0	0	0	0
m8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
m9	0	0	0	0	0	0	0	0	0	0	0	0	0	1
m10	0	0	0	0	0	0	0	0	0	0	1	0	0	0
m11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
m12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
m13	0	0	0	0	0	0	1	0	0	0	0	0	0	0
m14	0	2	0	0	0	0	0	0	0	0	0	0	0	0

La matrice ME résultant du système de la bibliothèque en ligne est construite dans la table 2. En appliquant l'algorithme de la figure 5 sur cette matrice ME, le résultat de notre analyse de l'ordre d'appels sur le système de la bibliothèque en ligne (illustré précédemment) est le suivant : les méthodes m14 et m2 constituent un modèle récurrent, car m14 est appelée avant m2 plus d'une fois, et les méthodes m14 et m2 appartiennent à deux classes différentes. Par conséquent, il existe un aspect enchevêtré de type before, dont la méthode ***m14*** est sa consigne, et la méthode ***m2*** est son point de coupure.

3.3. Filtrage des aspects candidats obtenus

La dispersion et l'enchevêtrement représentent les deux symptômes indiquant l'existence de la transversalité. La détection d'un symptôme sans l'autre affaiblit la précision.

L'utilisation unique de l'AFC pour détecter les préoccupations transversales semble être insuffisante, car elle ne prend pas en compte le symptôme de l'enchevêtrement. En outre, elle ne fait pas la distinction entre une consigne et un point de coupure, car elle considère les deux comme des aspects candidats. Or, une méthode appelée plusieurs fois peut être un point de coupure, et non pas la consigne de l'aspect.

Une méthode invoquée par un objet, dont son invocation est dispersée (en plusieurs fois dans des contextes différents), mais n'appartenant pas à un modèle récurrent de type *before* ou *after*, n'est pas considérée, par l'analyse de l'ordre d'appels, comme un aspect. Or, la modification de la manière d'invoquer cette méthode cause des changements coûteux dans les objets qui l'appellent. Par conséquent, l'analyse d'ordre d'appels ne prend pas en compte le symptôme de la dispersion.

Nous constatons que les deux analyses, l'analyse formelle des concepts et l'analyse de l'ordre d'appels présentées dans les sections 3.1 et 3.2 semblent être complémentaires. Leur utilisation simultanément afin de détecter les préoccupations dispersées et enchevêtrées augmente la précision, et diminue les faux aspects candidats.

A partir des résultats de l'application de l'AFC sur l'exemple de la bibliothèque en ligne, nous avons obtenu les méthodes suivantes comme des aspects candidats : ***{m2, m14, m3, m8}***. En appliquant l'analyse d'ordre d'appels, le seul aspect candidat obtenu est de type *before*, dont la méthode ***m14*** est sa consigne, et son point de coupure est méthode ***m2.***

Cependant, les méthodes ***m3*** et ***m8*** sont des méthodes dispersées, et la modification de leurs signatures nécessite des modifications dans les objets O2 et O6. En outre, la méthode ***m2*** identifiée par l'AFC comme aspect est un point de coupure de la consigne ***m14***. Afin de filtrer les résultats obtenus par les deux analyses, nous proposons le filtrage de la figure 6.

En appliquant l'algorithme du filtrage (figure 6) sur les aspects candidats obtenus par les deux approches : analyse formelle des concepts et analyse d'ordre d'appels, afin d'éliminer les faux aspects et de spécifier avec plus de précision la nature de la méthode détectée (point coupure ou consigne), nous constatons que la méthode ***m2*** obtenue par l'AFC comme une consigne est un point de coupure, et les méthodes ***m3*** et ***m8*** sont des consignes d'aspects candidats qui n'ont pas été détectés par l'analyse d'ordre d'appels.

Les aspects candidats finaux obtenus par notre approche sont ***les méthodes m3, m8*** et ***m14*** : `Login(), logout()` et `modifier_nb_exemplaire(dec: int)`.

- Si un aspect candidat dispersé est déjà identifié comme enchevêtré, alors l'aspect dispersé sera éliminé, et le symptôme de dispersion est ajouté à l'aspect enchevêtré.
- Soit Pt_i est l'ensemble des points de coupure de l'aspect candidat $aspect_i$, Ad_i est sa consigne et $type_i$ son type. Si il existe $aspect_1$ et $aspect_2$, tel que $Pt_1 = Ad_2$ and $Pt_2 = Ad_1$ et $type_1 \neq type_2$ alors l'aspect candidat dispersé sera éliminé.

Figure 6. *Algorithme du filtrage des aspects candidats*

4. Validation

4.1. Métrique de couplage

Afin d'évaluer notre approche proposée, et de détecter les erreurs possibles dans le processus d'identification d'aspects enchevêtrées, nous utilisons la métrique du couplage [3] entre les classes, que nous appelons *CM* (*Coupling Metric*). Elle est calculée pour chaque aspect candidat enchevêtré. *CM* mesure le degré du couplage de classes. Ce couplage est le résultat des relations de ces classes avec un aspect candidat enchevêtré, soit *Asp*.

$$CM\ (Asp) = |classes\ liées\ à\ Asp|$$

Tel que chaque classe de l'ensemble " $classes\ liées$ à Asp" est une classe où appartient une méthode de la consigne de *Asp* ou une méthode de son point de coupure. La valeur de *CM* doit être supérieure à 1. La valeur élevée de *CM* signifie que l'aspect candidat Asp a plus la chance d'être refactorisé. Dans la section suivante, nous allons appliquer la métrique *CM* sur les résultats de l'étude de cas choisie.

4.2. Etude de cas

L'étude de cas que nous avons choisie est une version simplifiée du système de péage des autoroutes portugaises (*The Portuguese Highways Toll system*) [25].

"Dans un système de tarification de la circulation, les conducteurs de véhicules autorisés avancent devant les barrières de péage automatique. Les ports sont placées sur des voies spéciales appelées voies vertes. Un conducteur doit installer un dispositif dans son véhicule appelé *gizmo*. L'enregistrement des véhicules autorisés inclut les données personnelles du propriétaire, son numéro du compte bancaire et les détails du véhicule. Le gizmo est donné au client pour être activé en utilisant un guichet automatique (*ATM*) qui informe le système lors de l'activation du gizmo. Un gizmo est lu par les capteurs de barrière du péage. Les informations lues sont stockées par le système, et utilisées pour débiter les comptes correspondants. Lorsqu'un véhicule autorisé traverse une voie verte, le feu vert est allumé, et le montant étant débité est affiché. Si un véhicule non autorisé passe, un feu jaune est allumé, et une caméra prend une photo de la plaque (pour trouver le propriétaire du véhicule). Il existe trois types de barrière de péage : péage unique où le même type du véhicule paye un montant fixe, péage d'entrée pour entrer dans un péage et péage de sortie pour le quitter. Le montant payé sur les autoroutes dépend du véhicule et la distance parcourue. " [25]

A partir du diagramme de classes modélisant le système de péage des autoroutes portugaises, les classes du système sont : $Véhicle$, $Compte$, $Gizmo$, Feu, $Montant$, $Caméra$, $Affichage$ et $Entrée$, et leurs méthodes sont les suivantes :

1 $Enregister_véhicule$

2 $Vérifier_compte$

3 $Activer_gizmo$

4 $Lire_gizmo$

5 $Vérifier_gizmo$

6 $Allumer_vert$

7 $Calculer_montant$

8 $Débiter$

9 $Afficher_montant$

10 $Prendre_photo$

11 $Enregistrer_passage$

12 $Vérifier_entrée$

13 $Changer_rouge$

Table 3. *Résultats de l'application de notre approche sur le système de péage des autoroutes portugaises*

Aspect candidat	Symptôme	Consigne	Point de coupure	Type	*CM*

1	Enchevêtrement	{ Vérifier_gizmo }	{ Allumer_vert }	before	2
2	Enchevêtrement	{ Vérifier_gizmo }	{ Allumer_rouge }	before	2
3	Enchevêtrement	{ Allumer_vert }	{ Calculer_montant }	before	2
4	Enchevêtrement	{ Afficher_montant }	{ Débiter }	after	2
5	Enchevêtrement	{ Allumer_rouge }	{ Prendre_photo }	before	2
Aspect candidat	**Symptôme**		**Méthode**		
6	Dispersion		{ Enregistrer_passage }		

A partir des diagrammes de séquence modélisant le système de péage des autoroutes portugaises, la table 3 présente les aspects candidats obtenus par notre approche. A partie de la tables 3, nous constatons ceci :

- ✓ Aspect 1: Avant de d'allumer le feu vert, il faut d'abord vérifier le gizmo.
- ✓ Aspect 2: Aspect 1: Avant de d'allumer le feu rouge, il faut d'abord vérifier le gizmo.
- ✓ Aspect 3: Avant de calculer le montant, le feu vert doit être allumé.
- ✓ Aspect 4: Après avoir débiter le montant, il faut l'afficher.
- ✓ Aspect 5: Avant de prendre photo de la plaque du véhicule, il faut que le feu rouge soit allumé.
- ✓ Aspect 6: La méthode $Enregistrer_passage$ est invoquée trois fois par deux classes différentes : Feu et $Caméra$. Par conséquent, c'est une méthode dispersée.

Les aspects identifiés par notre approche prennent en compte les contraintes de l'étude de cas, et améliorent la modularité du système logiciel. En appliquant la métrique CM (de la section 4.1), nous constatons dans la table 3 que chaque aspect candidat enchevêtré relie deux classes.

Cette étude de cas a été utilisée dans d'autres travaux, afin d'illustrer l'identification des préoccupations transversales. Par exemple, dans [26], les préoccupations non fonctionnelles sont considérées comme des préoccupations transversales candidates, et les aspects obtenus sont : Security, $Persistence$, $Data\ representation$ et $Event\ notification$. Dans [27], les aspects obtenus sont : $Response\ Time$, $Availability$, $Security$, $Legal\ Issues$, $Compatibility$, $Correctness$, $Multi-Access$. Les aspects détectées par ces approches sont généraux, initialement identifiés à partir des connaissances du domaine et ils peuvent ne pas apparaitre dans l'implémentation du système comme attendu.

5. Travaux connexes

Les préoccupations transversales recouvrent multiples phases du cycle de développement du logiciel. Afin de supporter une meilleure modularité des systèmes logiciels, en bénéficiant des avantages du paradigme orienté aspect, plusieurs approches visent à identifier ces préoccupations transversales, afin de permettre leur modularisation dans des aspects. Ces approches opèrent sur différentes phases du cycle du développement logiciel. Nous pouvons classer ces approches en deux grandes familles.

La première famille vise une détection précoce des aspects durant les premières phases du développement. Les approches de cette famille visent à permettre un développement orienté aspect, par une gestion précoce de la transversalité [14]. Ces approches souffrent de certaines limites [1]. Les approches basées sur les interviews avec les intervenants sont souvent imprécises, pleines de contradictions et manquent de l'information essentielle. En outre, les préoccupations transversales détectées tôt dans le cycle du développement peuvent être changées dans les phases ultérieures : disparaissent ou émergent.

La deuxième famille des approches d'identification d'aspects s'accentue sur la détection des préoccupations à partir du code source [11]. Plusieurs approches se sont basées sur les méthodes du système logiciel afin de détecter la transversalité au niveau du code source. Parmi ces approches, il existe ceux qui regroupent les méthodes selon un critère défini. L'analyse formelle des concepts [2] et la technique du clustering [15] sont deux techniques utilisées pour regrouper les méthodes du système logiciel selon leurs invocations. La méthode qui contient des appels appartenant à différentes classes est vue comme un aspect candidat. Notre approche utilise l'AFC dans ce sens, or ces approches [2, 15] opèrent au niveau code source et détectent seulement le symptôme de dispersion.

L'analyse AFC a été également utilisée sur les traces d'exécution [20] afin de regrouper les méthodes selon les cas d'utilisation qui les invoquent durant la génération des traces d'exécution. Notre approche a utilisé l'AFC afin de regrouper les méthodes selon les objets qui les invoquent durant les transmissions des messages des diagrammes de séquence.

L'analyse fan-in est une technique d'aspect mining (extraction d'aspects) [7] qui exploite les appels des méthodes. Elle détermine le degré de dispersion du code, en se basant sur le nombre de toutes les invocations d'une méthode (métrique *fan-in*) dans le système logiciel. La méthode dont sa métrique fan-in est supérieure à certain seuil est une préoccupation transversale. Notre approche utilise la métrique *fan-in* afin d'identifier seulement la dispersion (méthodes ayant *fan-in* > 1).

Certains travaux proposent [21] l'utilisation d'une mesure de distance qui est une fonction d'agrégation des symptômes de dispersion, clonage du code (*code cloning*) et convention de noms, utilisée pour regrouper les méthodes. La distance de dispersion entre deux méthodes est réduite lorsque l'ensemble de leurs classes et les méthodes qui les invoquent est plus grand. La distance de l'opération de clonage entre deux méthodes est réduite lorsque l'ensemble de leurs méthodes et les classes invoquées est plus grand. La distance de nom capture l'utilisation des conventions dans les noms de méthodes, classes, types de retour et de paramètres. Cependant, les distances définies dans cette approche sont imprécises, car dans la première distance, l'existence d'une méthode invoquée par la classe où la seconde méthode est définie, ou l'existence de deux méthodes invoquées par la même classe n'indique pas nécessairement que ces deux méthodes appartiennent au même aspect dispersé. Dans la seconde distance, les mêmes méthodes ou classes invoquées par deux méthodes n'indiquent pas nécessairement que ces deux méthodes appartiennent au même aspect de clonage. Dans la troisième distance, l'auteur se base sur l'analyse syntaxique qui est imprécise, car elle dépend de la syntaxe du programmeur. En outre, l'approche ne détecte pas le symptôme d'enchevêtrement, car avant d'appliquer l'algorithme du clustering, elle élimine les méthodes dont leur *fan-in* est inférieur à 2 (or ces méthodes peuvent être enchevêtrées).

Une autre approche qui exploite les appels entre les méthodes est représentée dans [22]. L'auteur a défini deux types de transversalité. Celle statique est due aux déclarations (dispersées et enchevêtrées) des méthodes et types substituables qui sont générés grâce aux relations statiques (*inheritance, implementation* ou *containment*) [23]. Avec cette transversalité statique, l'auteur définit les préoccupations du système. La transversalité dynamique est introduite grâce aux invocations dispersées et enchevêtrées des méthodes. En effet, une invocation d'une méthode affecte le flux du contrôle d'exécution en ajoutant

le comportement de la méthode appelée à celle appelante. Par conséquent, lorsqu'une méthode d'une préoccupation identifiée est invoquée par plusieurs autres méthodes appartenant à différentes autres préoccupations, le comportement de la préoccupation invoquée est dynamiquement dispersé (à l'exécution) avec les préoccupations associées aux appelantes. De la même façon, lorsqu'une méthode fait appels à d'autres méthodes appartenant aux différentes préoccupations, les comportements de telles préoccupations sont enchevêtrés ensemble dans la méthode appelante. Cette approche n'est pas complète, car les préoccupations transversales qui n'ont pas une structure statique (en termes de modules du système) ne sont pas considérées.

La technique des modèles récurrents a été appliquée aux traces d'exécution du code source afin de détecter la transversalité [8]. Une succession des méthodes invoquées qui appartient à un modèle récurrent présente une préoccupation transversale enchevêtrée. Cependant, une méthode dont son invocation est dispersée (plusieurs fois dans différents contextes) qui n'appartient pas à un modèle récurrent, ne sera pas considérée, par cette technique, comme un aspect, pourtant le changement de la manière de son invocation peut entrainer des changements coûteux dans les objets qui l'invoquent. Notre approche utilise cette technique afin d'identifier l'enchevêtrement.

Les auteurs dans [24] proposent une approche d'aspect mining en utilisant l'arbre d'appels de méthodes, afin de dépasser les limites de l'approche dans [8] qui est incomplète, car la trace du programme contient seulement un ensemble particulier des entrées du programme. Une analyse dynamique complète est impossible pour exécuter tous les chemins possibles, ce qui diminue les aspects candidats. L'idée de base dans [24] est d'observer le code source et créer l'arbre d'appels de méthodes, dans le but d'obtenir les traces d'appels des méthodes du système logiciel. L'arbre d'appels de méthodes est un arbre binaire qui décrit les relations d'appels de méthodes. Il contient des nœuds de contrôle et des nœuds d'appels de méthode. Le nœud de contrôle marque le type d'expression (*sentence*) où la méthode est appelée, incluant le nœud de contrôle de la séquence du nœud de contrôle, le code de contrôle if (*if control code*), le nœud de contrôle *if-else*, le nœud de contrôle *switch* et le nœud de contrôle de boucle. Le nœud d'appel de méthode marque la méthode appelée. Les traces sont ensuite étudiées pour les relations d'appels de méthodes. Ces dernières peuvent être décrites par les relations *inside* et *outside* (comme dans [8]). Les relations d'appels récurrents de méthodes sont des préoccupations transversales potentielles, qui décrivent les fonctionnalités récurrentes dans le programme, et par conséquent sont des aspects possibles. Notre identification de l'enchevêtrement en utilisant les appels récurrents des méthodes à un niveau plus abstrait ressemble à cette approche [24], or cette dernière souffre du manque de détection de la dispersion.

L'identification des aspects à partir des diagrammes UML générés du code source a été discutée dans [11]. L'auteur propose d'intégrer les aspects dans les diagrammes de séquence et d'activités générés à partir du code source orienté objet. Concernant le diagramme de séquence, les préoccupations qui traversent les séquences des messages sont transversales. Afin de détecter ces préoccupations transversales, les patrons récurrents d'exécution générés à partir des traces d'exécution du programme orienté objet seront considérés comme des aspects, car ils décrivent les fonctionnalités dupliquées dans le programme. Pour intégrer l'aspect dans le diagramme de séquence, l'aspect (patron récurrent) doit être remodularisé en utilisant la structure composite de l'UML (collaboration), outre les stéréotypes *<<aspect>>*, *<<advice>>*, *<<call>>* et *<<crosscut>>*. Les préoccupations transversales du diagramme d'activités sont celles qui traversent multiples procédures ou threads [11]. Les préoccupations transversales les plus importantes peuvent être des opérations de synchronisation ou d'exclusion mutuelle. Ensuite, grâce à la fusion et l'ajout des nœuds dans le diagramme d'activités, ce dernier

sera orienté aspect. Cependant, le seul symptôme de la transversalité détecté est l'enchevêtrement.

6. Conclusion

Dans ce papier, Nous avons proposé une approche d'identification des préoccupations transversales au niveau conceptuel, matérialisé par les diagrammes UML de classes et de séquences. Nous visons la détection des deux symptômes de la transversalité, par l'utilisation d'une approche hybride combinant deux analyses : l'analyse formelle des concepts et l'analyse de l'ordre des appels, et de filtrer ensuite l'ensemble des aspects candidats obtenus par ces deux analyses, afin de garder ceux jugés pertinents. En se basant sur les diagrammes de séquence, nous exploitons d'une part les relations d'appels des méthodes et d'autre part l'ordre chronologique d'exécution des tâches du système.

La perspective ouverte par notre travail se situe au niveau de l'application de notre approche sur d'autres diagrammes conceptuels tels que le diagramme d'activité.

7. Bibliographie

[1] Dahi, F., Bounour, N., 2011, Etude critique des approches de découverte d'aspect à travers le cycle du développement de logiciels, *2nd International Conference on Complex Systems (CISC'11)*, Algeria, Jijel University.

[2] Dahi, F., Bounour, N., 2011, Identification d'aspects par l'analyse des concepts formels, *1st International Conference on Information Systems and Technologies (ICIST'11),* Algeria, Tebessa University.

[3] Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A., 2003, On the Reuse and Maintenance of Aspect-Oriented Software: an Assessment Framework, *Brazilian Symposium on Software Engineering (SBES)*, Manaus, Brazil.

[4] Pawlak, R., Retaillé, J.P., Seinturier, L., 2004, *Programmation orientée aspect pour Java/J2EE*, Eyrolles.

[5] Hürsch W., Lopes, C., 1995, *Separation of Concerns, Technical report by the College of Computer Science,* Northeastern University.

[6] Ganter, B., Wille, R., 1999, Formal Concept Analysis: Mathematical Foundations, *Springer-Verlag*.

[7] Marin, M., Deursen, A., Moonen, L., 2004, Identifying aspects using fan-in analysis, *11th Working Conference on Reverse Engineering, IEEE Computer Society*.

[8] Breu, S., Krinke, J., 2003, Aspect mining using dynamic analysis, *In GI-Software technik-Trends, Mitteilungen der Gesellschaft f ur Informatik,* volume 23, Bad Honnef, Germany, pp. 21–22.

[9] Bennett, K.H., Rajlich, V., 2000, Software maintenance and evolution*: a roadmap. ICSE - Future of SE Track* , pp. 73-87.

[10] Ciraci, S., Broek, P., 2006, Modelling Software Evolution using Algebraic Graph Rewriting. In *Workshop on Architecture-Centric Evolution, ACE 2006, 3-7 July 2006, Nantes, France.*

[11] Su, Y., Li, F., Hu, S., Chen, P., 2006, Aspect-oriented software reverse engineering, *Journal of Shanghai University,* volume 10, number 5.

[12] Deursen, A., Marin, M., Moonen, L. 2003, Aspect Mining and Refactoring, *1st International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, University of Waterloo.

[13] Masuhara, H., Kiczales, G., 2003, Modelling Crosscutting in Aspect-Oriented Mechanisms. *17th European Conference on Object Oriented Programming (ECOOP)*, Darmstadt (2003)

[14] Moreira, A. Araújo, J., 2011, The Need for Early Aspects, *LNCS 6491, Springer-Verlag Berlin Heidelberg* , pp. 386–407.

[15] Bouasla, L., Bounour, N., 2011, Identification d'aspects en utilisant le clustering, *2nd International Conference on Complex Systems (CISC'11)*, Jijel University, Algeria.

[16] Siau K., Cao, Q., 2001, Unified Modeling Language: A Complexity Analysis, *journal of database management,* Volume 12.

[17] Mens , K., Kellens, A., Krinke, J., 2008, Pitfalls in Aspect mining , *WCRE '08 Proceedings of the15th Working Conference on Reverse Engineering.*

[18] `http://www.eclipse.org/aspectj/doc/released/faq.html`, 17/11/2013

[19] Kiczales, G., 1997, Aspect-oriented Programming, *European Conference on object-oriented Programming (ECOOP)*, Finland. Springer-Verlag LNCS 1241.

[20] Tonella, P., Ceccato, M., 2004, Aspect mining through the formal concept analysis of execution traces, *11th Working Conference on Reverse Engineering, IEEE Computer Society,* pp. 112–121.

[21] Fillus, E.K., Vergilio, S.R., 2012, A Clustering Based Approach for Aspect Mining and Pointcut Identification, *6th Latin American Workshop on Aspect Oriented Software Development Advanced Modularization Techniques,* Brazil.

[22] Bernardi, M.L., Lucca, G.A., 2011, Identifying the Crosscutting among Concerns by Methods' Calls Analysis, Vol. 257, pp. 147-158*, Springer.*

[23] Bernardi, M.L., Lucca, G.A., 2009, Analysing Object Type Hierarchies to Identify Crosscutting Concerns, *Springer-Verlag Berlin Heidelberg,* pp. 216–224.

[24] Qu, L., Liu, D., 2007, Aspect Mining Using Method Call Tree, *International Conference on Multimedia and Ubiquitous Engineering,* Korea.

[25] Clark, R., Moreira, A., 1997, Constructing Formal Specifications from Informal Requirements, *In: Software Technology and Engineering Practice*, pp. 68–75. IEEE Computer Society Press, Los Alamitos.

[26] Rashid, A., Moreira, A., Araujo, J., 2003, Modularisation and Composition of Aspectual Requirements. *2nd Aspect Oriented Software Conference* (AOSD), Boston, USA.

[27] Conejero, J.M., Hernándeza, J., Jurado. E., Berg, K., Early Aspect Mining in the Portuguese Highways Toll System, `http://www.google.fr/url?sa=t&rct=j&q=early%20aspect%20mining%20in%20the%20portuguese%20highways%20toll&source=web&cd=1&ved=0CDMQFjAA&url=http%3A%2F%2Fquercusseg.unex.es%2Fchemacm%2Fresearch%2Fanalysisofcrosscutting%2F%3Fdownload%3DPortugueseHighwaysTollSystem.pdf&ei=ov2fUY7FF8iw0AXehYCADg&usg=AFQjCNGA-ZK9DW_LwjZ6W_--r9m0ckZJJQ` , 11/11/2013