UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES
ET INFORMATIQUE APPLIQUÉES

PAR
YVES CÔTÉ

TESTS D'INTÉGRATION DANS LES SYSTÈMES ORIENTÉS ASPECT

DÉCEMBRE 2009

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

TESTS D'INTÉGRATION
DANS LES SYSTÈMES ORIENTÉS ASPECT

SOMMAIRE

Dans un système orienté aspect, les règles d'intégration entre aspects et classes sont définies dans les aspects, à l'insu des classes. L'aspect décrit, à l'aide de diverses constructions, comment cette intégration est effectuée. Ce niveau supplémentaire d'abstraction, ainsi que ses conséquences en termes de contrôle et de complexité doivent être pris en compte afin de s'assurer que les dépendances entre les aspects et les classes, en plus de celles qui existent entre les classes, soient testées adéquatement.

Les approches existantes pour le test de logiciel ne couvrent pas les spécificités du paradigme aspect. Le test orienté aspect apporte de nouveaux défis. De nouvelles stratégies de tests d'intégration doivent donc être développées. Durant le processus d'intégration, de nouveaux critères, en plus de ceux spécifiés pour le paradigme objet, doivent donc être pris en compte (intégration de un ou plusieurs aspects à une classe, complexité des aspects, types de dépendance, etc.).

Dans ce contexte, nous proposons une stratégie de tests d'intégration pour les systèmes orientés aspect basée sur les dépendances entre classes et aspects. Il s'agit, en fait, d'une extension d'une stratégie proposée pour les systèmes orientés objet. Cette nouvelle approche intègre des considérations de haut niveau (relation d'héritage entre classes, etc.) et des relations de conception détaillée représentant les multiples interactions qui existent entre les objets et les aspects. L'objectif principal consiste à mieux orienter le processus d'intégration des classes tenant compte des aspects, et de l'optimiser en réduisant le nombre de bouchons de test.

INTEGRATION TESTING IN ASPECT-ORIENTED SYSTEMS

ABSTRACT

In aspect-oriented programs, rules of integration between aspects and classes are defined in aspects without the knowledge of classes. Aspects describe, using various constructions, how this integration is carried out. This additional level of abstraction and its consequences in terms of control and complexity must be taken into account to ensure that dependencies between aspects and classes, in addition to those existing between classes, are tested adequately.

Existing object-oriented testing techniques do not cover the new dimensions introduced by the aspect paradigm. Aspect-oriented testing brings new challenges. New integration testing strategies must be developed. During the process of integration testing, new criteria in addition to those defined for object paradigm must be taken into account (integration of one or more aspects to a given class, complexity of aspects, types of dependencies, etc.).

In this context, we propose an integration testing strategy for aspect-oriented systems based on the dependencies between classes and aspects. It is, in fact, an extension of an existing strategy for object-oriented systems. This new approach incorporates high level considerations (inheritance relationship between classes, etc.) and detailed design relationships representing the multiple interactions between objects and aspects. The main objective is to better guide the integration of classes taking into account aspects, and to optimize it by reducing the number of test stubs.

REMERCIEMENTS

Je tiens personnellement à dédier ce mémoire de maîtrise à ma petite famille, en particulier à mon épouse Lucie ainsi qu'à mes deux enfants Andréane et Alexandre pour leur encouragement tout au long de ce processus.

Un gros merci à plusieurs membres de ma famille pour leur grande compassion même si le sujet de maîtrise ne leur semblait pas toujours évident.

À mes beaux parents qui, malgré leur âge respectable, m'ont supporté avec leur compréhension, signe d'une grande sagesse.

Il ne faudrait surtout pas oublier mes deux directeurs de recherche Linda Badri et Mourad Badri pour leurs conseils judicieux et leur rigueur sans quoi ce beau projet n'aurait pas pu se réaliser.

TABLE DES MATIÈRES

LISTE DES FIGURES

CHAPITRE 1

INTRODUCTION

1.1 Problématique

La programmation orientée objet éprouve certaines limites quant à la représentation des préoccupations transverses dans un programme. Le code correspondant à ces préoccupations est souvent dupliqué et dispersé dans les programmes, ce qui les rend difficiles à comprendre, à tester, à réutiliser et à maintenir. Le développement orienté aspect offre de nouvelles perspectives permettant une meilleure séparation des préoccupations transverses. Il permet de supporter une bonne factorisation de ces différentes préoccupations en les regroupant dans des unités modulaires appelées aspects. Ceci conduit à une réduction de la dispersion du code et une amélioration de sa modularité. Malgré les nombreux avantages que procure le paradigme aspect, il ne reste pas moins qu'il n'est pas encore mature. Il pose, en effet, plusieurs problèmes.

Les aspects apportent de nouvelles abstractions et dimensions en termes de contrôle et de complexité. Les approches existantes pour le test orienté objet ne couvrent pas les spécificités du paradigme aspect. Le test orienté aspect constitue donc un défi important. Le développement du paradigme aspect ainsi que sa généralisation sont liés, entre autres, au développement de techniques et d'outils supportant le test des programmes orientés aspect à différents niveaux. Les méthodes de test orientées objet actuelles, en particulier celles relatives aux tests d'intégration, ne sont pas adaptées pour la technologie aspect. Le code des aspects ainsi que les mécanismes permettant son

intégration au code objet constituent une nouvelle source de fautes. La relation entre les aspects et les classes diffère fondamentalement de celle présente entre les classes dans un système orienté objet.

La principale problématique vient de la relation entre les aspects et les classes. Les liens reliant un aspect à une classe ne peuvent être identifiés en analysant les classes. Une des formes majeures de dépendances entre les aspects et les classes vient du fait que le concept de l'appelant et de l'appelé, connu dans les systèmes orientés objet, prend dans les programmes orientés aspect un nouveau sens. La plupart des stratégies de test orienté objet se basent sur ce genre de relation entre les classes. Dans un système orienté objet, l'appelant et l'appelé, à un haut niveau, sont des classes. Dans ce type de système, un appelant spécifie les différents appels qu'il effectue ainsi que le contrôle lié à ces appels. Dans un système orienté aspect, l'inverse se produit puisque les règles d'intégration sont définies dans les aspects à l'insu des classes. La simple analyse des classes ne permet pas de savoir quels sont les aspects qui vont s'y greffer. L'aspect décrit, en fait, à l'aide de diverses constructions, comment cette intégration sera effectuée. Ce niveau supplémentaire d'abstraction, ainsi que ses conséquences en termes de contrôle et de complexité, doivent être pris en compte afin de s'assurer que les dépendances, entre les aspects et les classes (en plus de celles qui existent entre les classes), soient testées adéquatement. Durant le processus des tests d'intégration, de nouveaux critères, en plus de ceux spécifiés pour le paradigme objet [Bad 05], doivent donc être pris en compte (intégration de un ou plusieurs aspects à une classe, complexité des aspects, dépendances entre aspects, etc.).

1.2 Approche

De nouvelles stratégies de tests d'intégration doivent donc être développées pour les systèmes orientés aspect, tenant compte du niveau d'abstraction supplémentaire apporté par les aspects en plus des diverses dépendances qui existent déjà entre les classes (interactions entre classes, cycles, etc.). Plusieurs approches ont été proposées

dans la littérature pour résoudre les divers problèmes liés aux tests d'intégration des classes dans les systèmes orientés objet [Tai 99, Bri 02, Tra 00, Bad 05]. La stratégie de tests d'intégration proposée par Badri et al. [Bad 05] est basée sur une forme étendue d'un modèle de dépendance objet. Cette technique a été évaluée et comparée à d'autres techniques de tests d'intégration pour les systèmes orientés objet. L'étude expérimentale menée sur plusieurs projets a montré les avantages qu'elle procure [Bad 05].

Dans le domaine aspect, très peu de travaux ont porté sur cette problématique [Cec 05, Reg 07]. Dans ce contexte, nous proposons une stratégie de tests d'intégration pour les systèmes orientés aspect basée sur les dépendances entre classes et aspects (différents niveaux). Il s'agit, en fait, d'une extension de la stratégie proposée par Badri et al. [Bad 05] pour les systèmes orientés objet. Cette approche est basée sur un modèle qui dérive du diagramme de classes de conception et des diagrammes de collaboration UML. Elle intègre donc des considérations de haut niveau (relation d'héritage entre classes, etc.) et des relations de conception détaillée représentant les multiples interactions qui existent entre les objets et les aspects. L'objectif principal consiste à mieux orienter le processus d'intégration des classes (tenant compte des aspects) et de l'optimiser en réduisant le nombre de bouchons. La stratégie que nous proposons est incrémentale.

1.3 Plan du mémoire

Le mémoire est structuré en 7 chapitres. Le chapitre 2 traite de la technologie aspect et du langage AspectJ. Les principaux concepts que véhicule ce paradigme y sont présentés. Une brève introduction au langage AspectJ est également proposée. Le chapitre 3 comporte plusieurs volets. On commence par présenter les différents types de dépendances (classe-aspect) et les principales mesures de couplages. Ces dépendances et ces mesures de couplages entrent dans la définition des principaux critères que nous avons considérés au niveau de notre approche. Ce contenu permettra de clarifier, en particulier, les différents liens qui existent entre les aspects et les classes. On présentera,

également, la notion de bouchon de test qui constitue un élément essentiel pour l'intégration. Les stratégies de tests d'intégration seront introduites en commençant par celles développées dans le contexte du paradigme orienté objet. Par la suite, nous présenterons les quelques travaux effectués sur les tests d'intégration dans le contexte du paradigme aspect. Le chapitre 4 portera sur la présentation de l'algorithme B3 proposé par Badri et al. [Bad 05] pour les systèmes orientés objet. Les étapes fondamentales de l'algorithme y seront élaborées. Le chapitre 5 nous amènera au cœur de notre sujet. Il portera sur l'extension réalisée de l'algorithme B3 pour les systèmes orientés aspect. Les principales étapes de la démarche adoptée y sont décrites et illustrées. Les mécanismes d'interactions et de couplage, entre autres, seront abordés. Le chapitre 6 présente l'étude de cas réalisée afin de démontrer la faisabilité de notre approche et d'en effectuer une première évaluation. Le chapitre 7 présente les conclusions générales de notre travail.

CHAPITRE 2

TECHNOLOGIE ORIENTÉE ASPECT

2.1 Concepts orientés Aspect - AspectJ

AspectJ, le langage de programmation orientée aspect le plus abouti, est issu d'un projet très ambitieux. La programmation orientée aspect est une technologie récente qui vise l'amélioration de la séparation des préoccupations transversales (crosscutting concerns) en introduisant une unité spéciale appelée aspect. Le projet Aosd [Aos 05] ayant conduit à la création du langage AspectJ a pendant longtemps existé de manière autonome. Aujourd'hui, il est supporté par la communauté Eclipse.org [Ajp 02] qui fournit la plate-forme et plusieurs utilitaires. Le langage AspectJ [Lad 02] comporte plusieurs notions qui sont:

L'advice : un *advice* est un mécanisme (similaire à une méthode) utilisé pour spécifier le code à exécuter lors de la capture d'un point de jointure par le programme. L'avantage des *advice* vient du fait qu'ils peuvent avoir accès à certaines valeurs du contexte d'exécution d'un point de coupure. Les points de coupure et les *advice* définissent à eux deux les règles d'intégration.

Le point de jointure (joinpoint) : il représente des balises dans le code du programme où s'exécute un aspect. Les points de jointure ont comme principale tâche de définir la structure des préoccupations que nous désirons factoriser en désignant des points bien précis dans l'exécution d'un programme. AspectJ permet, entre autres, de définir les points de jointure relatifs à un appel de méthode ou de constructeur. Il est possible de

regrouper plusieurs points de jointure à l'aide des points de coupure.

Le point de coupure (pointcut) : un point de coupure définit un ou plusieurs points de jointure. Les points de coupures servent à définir le champ d'action des advice qui leurs sont associés. Les points de coupure sont composés d'une série de spécifications qui définissent l'ensemble des points de jointure qu'ils contiennent. Un pointcut permet, par exemple, d'identifier l'invocation d'une méthode particulière appartenant à une classe bien précise.

Aspect : un aspect est une unité de regroupement de définitions de points de coupure nommés, d'associations de points de coupures à des *Advice*, et d'introductions d'attributs ou de méthodes dans les classes cibles.

Weaver (tisseur) **:** le weaver permet d'intégrer le code aspect au code de base. Il existe, cependant, deux types de tisseurs [Vie 05] :

- Tisseur dynamique : un tisseur dynamique tisse tous les aspects dynamiquement au moment de l'exécution du programme. Ces tisseurs doivent être très rapides pour ne pas ralentir l'application.

- Tisseur statique : un tisseur statique tisse les aspects au moment de la compilation directement dans le code des classes. Les tisseurs statiques ont moins de contraintes de temps d'exécution que les tisseurs dynamiques.

Les principaux tisseurs existants ont été développés pour Java. Parmi eux, on y retrouve AspectJ et JAC (Java Aspect Component). AspectJ est un tisseur statique très puissant, qui génère soit un fichier source Java ou tout simplement un fichier bytecode.class. AspectJ est parfaitement intégré à la plate-forme Eclipse, grâce au plug-in AJDT.

La figure 1 donne un aperçu de l'interface de tissage AspectJ sous le module « visualiser ». Elle montre les zones de tissage sur les classes de l'application.

Figure 1 Tissage des aspects sur les classes

Le principe de base du fonctionnement d'AspectJ est très simple. En fait, tout se passe à la compilation du projet. L'ordre de compilation se fait comme suit :

- Le tisseur AspectJ recherche l'ensemble des définitions d'aspects à travers le projet, et les compile. Plus précisément, il précompile chaque aspect en une classe Java standard qui à son tour est compilée en bytecode.
- Si tous les aspects sont exempts d'erreurs de compilation, ils sont appliqués aux classes de base. Cela revient à identifier touts les points d'interceptions dans le code source des classes ciblées et à y insérer le code des advices associés.
- Une fois le code Java modifié, il est compilé en bytecode .

Les aspects factorisent le comportement de certaines classes. Lors du tissage (weaving), ils sont greffés au code objet. La notion d'aspect est le cœur central du paradigme aspect. Il prend en compte les règles de tissage et permet de modifier le comportement d'une classe en particulier, et ce dans un contexte de préoccupation transversale. Les aspects sont similaires aux classes, avec néanmoins certaines différences importantes [Lad 03], entre autres:

- Les aspects peuvent inclure des données membres et des méthodes aux classes.

- Les aspects peuvent être déclarés eux-mêmes abstraits et doivent être développés.

- Les aspects doivent être greffés à l'intérieur des classes.

- Les aspects ne peuvent pas être instanciés. En d'autres mots, nous ne pouvons utiliser « new » en vue de créer un objet pour un aspect.

- Les aspects peuvent hériter des classes et d'aspects abstraits et ne peuvent hériter d'aspects concrets. Ils peuvent également implémenter des interfaces.

2.2 Objectifs de la technologie aspect

L'objectif de la technologie aspect est de permettre une « modularisation » du code correspondant à une préoccupation dispersée dans diverses classes (croscuting concern). Un aspect doit être un module dont le code [Klm 97] pourra être dispersé de façon automatique par le système. Cette dispersion est ainsi évitée par le programmeur qui localise, avec l'aspect, le code relatif à une certaine fonctionnalité ou propriété du programme à un seul endroit, l'aspect. Ainsi, la programmation [Lad 02, Lad 03, Xat 03] par aspect veut éviter d'avoir du code dispersé (scattering concern) ou bien du code

entremêlé (tangling concern).

Scattering : Il s'agit de code similaire qui est distribué dans plusieurs classes ou parties différentes du programme. Il peut y avoir inconsistance entre les pièces de code, car l'une d'elles peut être utilisée ou modifiée indépendamment des autres alors que leur fonctionnement doit être identique.

Tangling : Plusieurs modules dans un système peuvent interagir simultanément les uns avec les autres sous certaines conditions. Cette multitude de conditions a pour conséquence un enchevêtrement de code.

2.3 Étapes du développement orienté aspect

Dans le processus de développement du logiciel [Bal 02], l'implémentation d'une application exige certaines étapes :

- La décomposition des éléments du système (classes, aspects) - on sépare toutes les préoccupations qu'elles soient fonctionnelles ou non.

- L'implémentation de chaque préoccupation transverse sera codée séparément dans un aspect. Pour un aspect donné, le programmeur fixe les règles d'intégration de l'aspect avec le reste du système.

- L'intégration du système : le langage orienté aspect offre un mécanisme d'intégration appelé « Weaver ». Celui-ci va donc composer le système final sur la base des règles qui lui ont été dictées. On peut voir à la figure 2 le résultat d'une intégration en vue du système final.

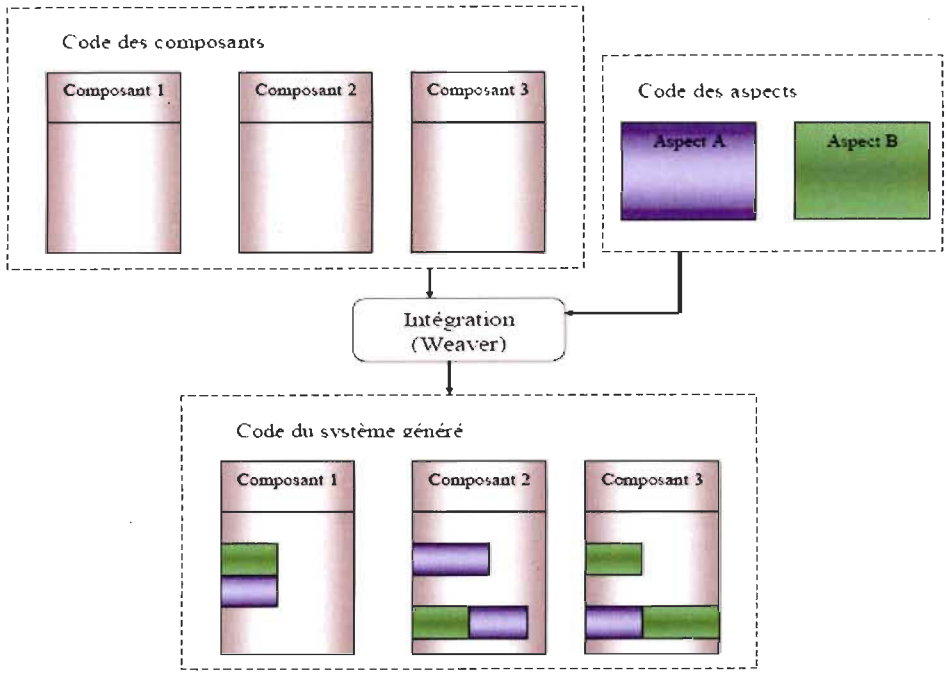


Figure 2 Intégration du code aspect en vue du système final

2.4 Avantages

Les points positifs résident dans la richesse du langage, en particulier dans la description des points de coupure. Par ailleurs, la « modularisation » du code relatif aux préoccupations transverses permet de les écrire et de les maintenir de façon isolée [Bal 02], évitant ainsi, la duplication de code. Les systèmes orientés aspect [Bal 02] sont très évolutifs, car les classes ne sont pas au courant des problématiques qui les recoupent. De ce point de vue, il devient plus simple d'ajouter des nouvelles préoccupations. En effet, dans le cas où on veut ajouter de nouvelles exigences non-fonctionnelles comme la synchronisation, il s'agit de créer un nouvel aspect qui s'en occupera sans toucher au code des composantes existantes. Par ailleurs, une étude [Wal 99] a permis de montrer que les programmeurs mettent beaucoup moins de temps à « débogger » le code source grâce au langage orienté aspect. Ceci nécessite néanmoins d'autres investigations pour pouvoir tirer des conclusions finales.

2.5 Impact du code non modularisé

A partir des notions évoquées précédemment [Lad 02] et des concepts de « tangling » et « scattering », cette façon de faire (structuration du programme en classes et aspects) peut avoir un impact significatif au niveau du design et du développement des programmes. Même si chaque item est traité séparément, ils ont pourtant des impacts les uns sur les autres. On peut avoir en particulier [Lad 02]:

- *Un traçage difficile* : les différentes préoccupations d'un logiciel deviennent difficilement identifiables dans l'implémentation. Il peut en résulter une correspondance assez obscure entre les exigences et les implémentations.

- *Une diminution de la productivité* : l'implémentation simultanée de plusieurs préoccupations met l'emphase sur plusieurs préoccupations dites périphériques au lieu de la préoccupation principale, ce qui a pour impact pour le développeur de s'éloigner de l'objectif principal.

- *Une diminution de la qualité du code* : les programmeurs ne peuvent pas se concentrer sur plusieurs contraintes à la fois. L'implémentation disparate de certaines préoccupations peut entrainer des effets de bords non-désirés.

Même si les aspects semblent donner plus de souplesse dans le développement, ils peuvent causer des problèmes, et parfois de façon insoupçonnée. Les aspects peuvent interagir entre eux de façon indirecte et créer des effets de bord (conflits entre aspects). De plus, les aspects ont la capacité de changer la sémantique des préoccupations [Mce 05]. Lorsque les aspects s'intègrent aux classes [Han 03], l'utilisation ponctuelle des joint point nous amène deux types de problématique soit :

- Point de jointure de type accidentelle : l'utilisation de token <<*>> peut nous amener à une certaine confusion.

- Récursivité accidentelle : multitude d'appels récursifs lorsque le point de jointure est inclus dans le code de l'aspect.

Conflits Aspect-Aspect

Exécution conditionnelle : le fonctionnement d'un aspect est conditionnel à l'exécution d'un autre.

Exclusion mutuelle : le fonctionnement d'un aspect exige l'exclusion d'un autre.

Ordonnancement : séquencement précis (l'un à la suite de l'autre) si deux aspects utilisent le même point de jointure.

Ordonnancement dit dynamique : dépend du contexte dynamique dans lequel les aspects s'exécutent.

Changement de certaines fonctionnalités : le fait d'ajouter un nouvel aspect peut empêcher un autre aspect d'atteindre un point de jointure.

Comportement inconsistant : se produit lorsqu'un aspect détruit l'état d'un autre aspect ou une préoccupation de base.

Anomalies de composition : par exemple, un aspect altère ou substitue un élément qu'il partage avec un autre aspect concurrent.

Conflit Base-Aspect

Cette problématique se produit lorsque nous sommes en présence d'un aspect invasif. On notera une dépendance circulaire entre la base et l'aspect. Un aspect est considéré comme invasif s'il manipule le flot de données ou le flot de contrôle du programme de base. Il existe une liste de structures invasives; ces structures peuvent être implémentées par un aspect ou par un greffon. Parmi ces structures invasives on peut citer :

Hiérarchie - L'aspect modifie la hiérarchie des classes, par exemple en ajoutant un parent à une classe existante.
Ajout d'un attribut - L'aspect ajoute un nouvel attribut à une classe existante.

Une autre problématique se situe au niveau de la lisibilité : Il est difficile de voir de façon claire comment les aspects s'appliquent au programme. Il faut pour cela lire chacun des points de coupure des aspects, puis lire le programme afin de découvrir les correspondances. Même si l'implémentation de l'aspect est modularisée, la définition de son application requiert la connaissance de tout le programme et des points de jointure qui le compose. Plusieurs auteurs ont étudié d'autres volets concernant les conflits pouvant survenir sournoisement. Citons Tessier et al. [Tes 04] qui ont identifié différents conflits tels que la présence de *joint point* accidentels, de dépendance circulaire et conflits provenant de différentes préoccupations transversales issues d'interactions inattendues et non désirables. Balzarotti [Bal 04] s'est intéressé à comment une modification d'un module peut altérer le comportement d'un système. Havinga [Hav 07] propose une modélisation basée sur des graphes en présence d'aspects. A travers des règles précises, il vérifie les transformations pouvant survenir au niveau du code source. Lagaisse [Lag 04] propose une extension du paradigme de *design par contrat* où les aspects sont mis à contribution. Ceci permettra de déterminer quel sera l'impact de l'introduction de nouveaux aspects sur les aspects initiaux qui ont été tissés préalablement.

CHAPITRE 3

ÉTAT DE L'ART

3.1 Présentation générale

Malgré les nombreux avantages que procure le paradigme aspect, il ne reste pas moins qu'il n'est pas encore mature. Il pose plusieurs problèmes, en particulier pour le processus de test. Les aspects apportent de nouvelles abstractions et dimensions en termes de contrôle et de complexité. Les approches existantes pour le test de logiciel ne couvrent pas les spécificités du paradigme aspect [Ale 04, Bal 01, Mor 04]. Le test orienté aspect constitue donc un défi important. Les méthodes de test orienté objet actuelles, en particulier celles relatives aux tests d'intégration, ne sont pas comme mentionné précédemment adaptées pour la technologie aspect. Le code des aspects ainsi que les mécanismes permettant son intégration au code objet peuvent provoquer de nouvelles fautes [Ale 04]. La relation existante entre les aspects et les classes diffère de celle présente entre les classes dans un environnement orienté objet [Bal 01]. De nouvelles stratégies de tests d'intégration doivent donc être développées pour les systèmes orientés aspect, tenant compte du niveau d'abstraction supplémentaire et des diverses dépendances qui existent déjà entre les classes (interactions entre classes, cycles, ...).

La principale problématique vient de la relation entre les aspects et les classes. Les liens, reliant un aspect à une classe, ne peuvent être identifiés en analysant les classes [Ale 04, Bal 01, Mor 04]. Une des formes majeures de dépendances entre les aspects et les classes vient du fait que le concept de l'appelant et de l'appelé, connu dans les systèmes orientés objet, prend dans les programmes orientés aspect un nouveau sens [Bal 01]. La plupart des stratégies de test orienté objet se basent sur ce genre de

relation entre les classes. Dans un système traditionnel, un appelant spécifie les différents appels qu'il effectue ainsi que le contrôle lié à ces appels. Dans un système orienté aspect, l'inverse se produit puisque les règles d'intégration sont définies dans les aspects à l'insu des classes. L'aspect décrit, à l'aide de diverses constructions, comment cette intégration sera effectuée. Ce niveau supplémentaire d'abstraction, ainsi que ses conséquences en termes de contrôle, doivent être pris en compte afin de s'assurer que les dépendances, entre les aspects et les classes, soient testées adéquatement [Bal 01]. Durant le processus des tests d'intégration, de nouveaux critères, en plus de ceux spécifiés pour le paradigme objet [Bad 05] doivent donc être pris en compte (intégration de un ou plusieurs aspects à une classe, complexité des aspects, dépendance entre aspects, dépendance entre classe et aspect, ...).

Les tests d'intégration ont pour objectif de révéler les fautes qui entravent le bon fonctionnement des interactions entre les différents composants d'un système. Elles peuvent se situer au niveau fonctionnel ou au niveau structurel voire même une combinaison des deux [Mcg 94]. L'intégration des classes en vue de créer une application doit répondre à l'approche globale de développement. Selon Binder [Bin 94], ces tests possèdent deux stratégies :

- Stratégie *Thread-based* qui intègre plusieurs classes qui répondent aux mêmes stimulis.
- Stratégie *Used-based* utilisée pour un cas d'utilisation donné : les classes activées sont testées.

La problématique des stratégies d'intégration s'articule autour des dépendances dont le principal enjeu est rencontré lorsque les composants interagissent entre eux. La présence de cycles dans les systèmes engendre différentes relations conduisant à un couplage fort entre les composantes compliquant davantage le problème [Bad 04]. Le problème fondamental consiste à définir un ordre efficace d'intégration des composants. Il faudra trouver alors un ordre de séquencement pour l'intégration des composantes

[Mil 02] tenant compte des dépendances induites. Dans le processus d'intégration, plusieurs critères doivent être pris en compte. La complexité des interactions et les dépendances peuvent rendre le processus d'intégration ardu. On peut citer en l'occurrence, la complexité des composantes, les dépendances, le nombre de bouchons de test (réels et effectifs) etc. Par ailleurs, la présence d'aspects apporte aussi une autre forme de dépendance au sein d'un système. L'ordonnancement de l'intégration des composants avec les aspects devra être étudié avec rigueur.

3.2 Notion de dépendance

3.2.1 Dépendances générées par les aspects et les classes

Le Modèle de Dépendance entre Classes (MDC) est issu du diagramme de conception UML. La figure 3 illustre le concept de dépendance (noté U) entre classes. La classe A dépend de la classe B qui dépend de la classe C. La notion de dépendance entre classes peut être interprétée comme une relation d'héritage ou d'utilisation.

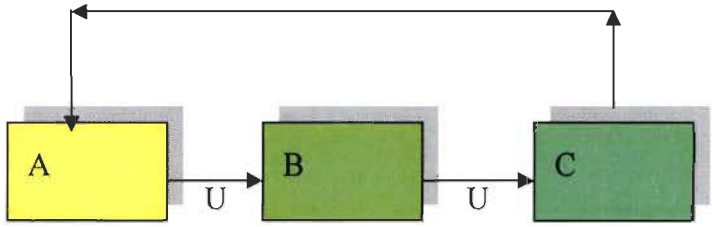


Figure 3 Modèle de dépendance entre classes (1)

La figure 3 illustre une forme de dépendance entre classes (relation d'utilisation). Il existe, cependant, d'autres formes de dépendance liée entre autres à la notion d'aspect. La figure 4 en donne un exemple.

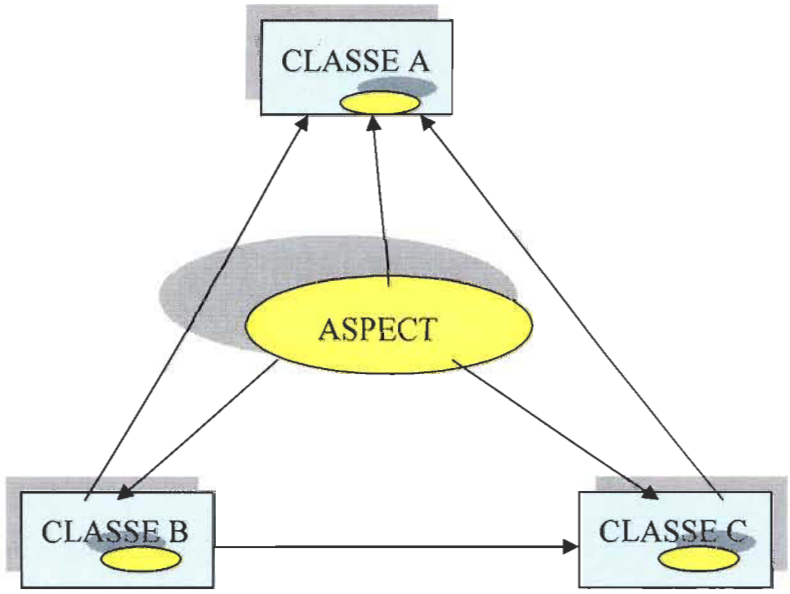


Figure 4 Modèle de dépendance entre classes et aspects (2)

Dans un système orienté objet [Mas 06], l'appelant et l'appelé, à un haut niveau sont effectivement des classes. Pour le modèle orienté aspect, ce concept doit être considéré différemment. Pour le cas qui nous intéresse, les règles d'intégration sont dictées par les aspects. Les classes n'ont pas le contrôle et ignorent la présence d'aspects. Cette approche apporte une nouvelle dimension à notre problème. Dès lors, il faut considérer les différents types de dépendance entre les aspects et les classes.

Selon [Reg 07], les aspects dépendent toujours des joinpoint définis dans les pointcuts. Les pointcuts sont définis par une combinaison de plusieurs éléments syntaxiques contenus dans leur déclaration. Pour identifier une relation de dépendance, on doit alors analyser tous les éléments dans la déclaration du pointcut. Dans un aspect, on peut avoir un ou plusieurs advices en liaison avec un ou *n* pointcut. On représentera par C dans la figure suivante la relation transverse « crosscuting ». On peut voir à la figure 5 [Reg 07] un modèle de dépendance transversale.

```
1 public class A{
2  public void print(){
3   System.out.println("A.print() invoked.");
4  }}
5 public aspect Aa {
6  pointcut printExecution():
7   execution (void A.print()):
8 }
```

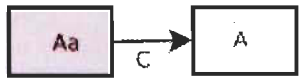


Figure 5 Dépendance transversale

Un autre type de dépendance [Reg 07] est celui généré par les relations entre advices et pointcuts au niveau aspects. On peut voir à la figure 6 suivante la relation entre un advice de type before de l'aspect Bb et le pointcut printExecution de l'aspect Aa où le poincut est utilisé à l'intérieur de l'advice. Cette dépendance est exprimée par « U » pour utilisation.

```
1 public aspect Aa{
2  pointcut printExecution():
3      execution (void A.print()),
4 }
5 public aspect Bb{
6  before(A ObjectA):Aa.printExecution(){
7   System.out.println("Before A.print() execution");
8  }}
```

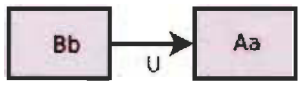


Figure 6 Dépendance de type utilisation

Des dépendances entre les aspects, où les pointcuts sont utilisés à l'intérieur d'autres pointcuts, peuvent parfois exister. La figure suivante [Reg 07] nous montre le pointcut printCallNotInA() qui utilise le pointcut printACall définissant une relation dénotée «U» entre les deux aspects Aa et Bb.

```
1 public aspect Aa{
2  pointcut printACall():call(void A.print());
3 }
4 public aspect Bb{
5   pointcut printCallNotInA():
6     Aa.printACall() && !within(A);
7 }
```

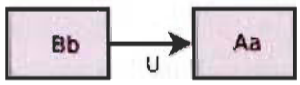


Figure 7 Dépendance entre pointcuts

Les informations capturées par les pointcuts grâce aux clauses this, target, args ou simplement l'instruction thisJoinPoint sont utilisées par les advices. Dans la figure 8 [Reg 07] l'advice before de l'aspect Aa utilise comme argument objectA, créant ainsi une dépendance associative.

```
1  public class A{
2   boolean checked=false;
3   public void print(){
4    System.out.println("A.print() invoked.");
5   }
6   public void hasChecked(){
7    checked=true;
8   }}
9  public class B{
10  public void print(A anObject){
11   System.out.println("B.print() invoked.");
12   anObject.print();
13  }}
14 public aspect Aa {
15  pointcut printExecution(A ObjectA):
16   execution (void B.print()) && args(ObjectA);
17  before(A ObjectA): printExecution(){
18   System.out.println("Before B.print() execution");
19   ObjectA.hasChecked();
20  }}
```

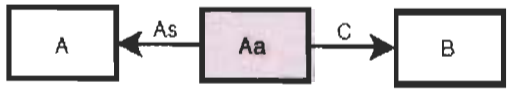


Figure 8 Dépendance associative

En AspectJ, il est possible d'utiliser une déclaration intertype afin de changer la structure de la classe par l'addition de méthodes et d'attributs. Ce type de déclaration peut, entre autre, modifier la structure d'héritage. L'introduction d'attributs à l'intérieur d'une classe peut dans certains cas être utilisée par une méthode de la classe en autant que l'attribut soit visible. Les méthodes introduites peuvent utiliser les attributs « privés » de la classe de base. Dès lors, une très forte relation est créée entre les classes de base et les aspects contenant ces relations intertypes. Ces aspects ne peuvent donc être testés indépendamment du code de base. Dans la figure 9 l'aspect Aa est en relation intertype avec la classe A qu'on nommera « It ».

```
1 public class A{
2  public void print(){
3   System.out.println("A.print() invoked.");
4  }}
5 public aspect Aa{
6  public void A.differentPrint(){
7   System.out.println("Different:");
8   this.print();
9  }}
```

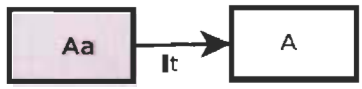


Figure 9 Dépendance intertype

Un aspect peut changer les propriétés d'héritage surtout s'il est associé à une ou plusieurs classes. Chaque clause *declare parents* de la figure 10 peut changer la relation d'héritage entre les deux classes visées [Reg 07]. Notons qu'un aspect, contenant ce type de clause peut seulement être testé, après l'implémentation des dites classes.

```
1  public class A{
2   public void print(){
3    System.out.println("A.print() invoked.");
4   }}
5  public class B{
6   public void print(){
7    super.print();
8    System.out.println("B.print() invoked.");
9   }}
10 aspect Aa{
11  declare parents: B extends A;
12 }
```

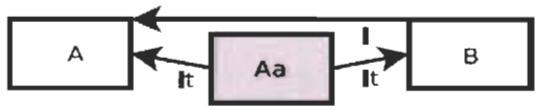


Figure 10 Dépendance intertype modifiant la hiérarchie pour l'héritage

En bref, les relations d'héritage entre les aspects et les classes font partie du type des dépendances (considérées comme interactions) dans un système orienté aspect. L'ordonnancement proposé par plusieurs auteurs, dans la manière d'aborder le test d'une manière globale des systèmes orientés aspect, repose tout d'abord sur le principe de tester les classes avant (et séparément dans un premier temps) les aspects. Par ailleurs, certains aspects dépendent d'autres aspects, et que cela peut engendrer des cycles de dépendance. Nous devons donc, à partir de ces modèles, adopter une stratégie pour l'ordonnancement de l'intégration des aspects et classes tenant compte de toutes ces dépendances et de leurs conséquences (les cycles que cela peut engendrer en particulier).

Par ailleurs, Zhao [Zha 04] a proposé plusieurs définitions (données dans ce qui suit) relatives au couplage dans un système orienté aspect. Ces mesures seront utilisées lors de l'intégration des aspects aux classes dans le cadre de la démarche que nous avons adoptée.

Définition 1 : Mesure de dépendance attribut-classe

Soit S un système OA (Orienté Aspect), a ε A(S) est un aspect de S et c ε C(S) est une classe de S, la mesure de dépendance attribut-classe s'exprime comme suit :

$$\delta_{AtC(a)} = \sum_{c \in C(S)} AtC(a,c)$$

Elle permet d'évaluer le nombre de dépendances AtC entre un aspect et quelques classes dans un système S.

Définition 2 : Mesure de dépendance module - classe

Dans le cas présent, la notion de module peut correspondre à un advice, un pointcut, une déclaration intertype ou bien une méthode.

Soit S un système OA, a ε A(S) est un aspect de S et c ε C(S) est une classe de S, la mesure de dépendance selon le cas peut s'exprimer sous différentes façons :

1) Dépendances advices-classes :

$$\delta_{AC(a)} = \sum_{c \in C(S)} AC(a,c)$$

Elle évalue le nombre de dépendances advices-classes entre un aspect a et quelques classes dans un système S.

2) Dépendance méthode-classe :

$$\delta_{MC(a)} = \sum_{c \in C(S)} MC(a,c)$$

Elle évalue le nombre de dépendances méthodes-classes entre un aspect a et quelques classes dans un système S.

3) Dépendance pointcut-classe :

$$\delta_{PC(a)} = \sum_{c \in C(S)} PC(a,c)$$

Elle évalue le nombre de dépendances pointcut-classes entre un aspect a et quelques classes dans un système S.

Définition 3 : Mesure de dépendance module-méthode

Soit S un système OA, a ε A(S) est un aspect de S et c ε C(S) est une classe de S, la mesure de dépendance module–méthode selon le cas peut s'exprimer sous différentes façons :

1) Dépendance advice-méthode:

$$\delta_{AM(a)} = \sum_{c \in C(S)} AM(a,c)$$

Elle évalue le nombre de dépendances advice-méthodes entre un aspect a et quelques classes dans un système S.

2) Dépendance méthode-méthode:

$$\delta_{MM(a)} = \sum_{c \in C(S)} MM(a,c)$$

Elle évalue le nombre de dépendances méthodes-méthodes entre un aspect a et quelques classes dans un système S.

3) Dépendance pointcut-méthode:

$$\delta_{PM(a)} = \sum_{c \in C(S)} PM(a,c)$$

Elle évalue le nombre de dépendances pointcut-méthodes entre un aspect a et quelques classes dans un système S.

Définition 4 : Mesure de dépendance aspect-héritage

Soit S un système AO, a ε A(S) est un aspect de S et c ε C(S) est une classe de S, la mesure de dépendance aspect-héritage s'exprime comme suit :

$$\delta_{AI(a)} = \sum_{c \in C(S)} AI(a,c)$$

Elle évalue le nombre de dépendances entre un aspect a et toutes les classes ancêtres de a.

Par ailleurs, Ceccato et Tonella [Cec 04] définissent un ensemble de 6 métriques de couplage orientées aspect. Bartsch et Harrisson [Bar 06] suggèrent aussi un cadre de travail basé sur les métriques (OO) préconisées par Briand et al. [Bri 99], mais adapté au modèle orienté aspect. Leur cadre de travail reprend celui évoqué en orienté objet, mais avec une approche différente. Ils suggèrent plusieurs mesures de couplage. La figure 11 visualise les différentes dépendances engendrées par les joinpoint [Bar 06].

#	Mechanism	Client Class	Client Item	Server Item	Server Class	Description
1	method execution join point	aspect a	advice b with pointcut c	join point d	class e	c picks out join point d associated with e
2	method call join point	aspect a	advice b with pointcut c	join point d	class e, class f	c picks out join point d associated with e and f
3	constructor call join point	aspect a	advice b with pointcut c	join point d	class e	c picks out join point d associated with e
4	constructor execution join point	aspect a	advice b with pointcut c	join point d	class e	c picks out join point d associated with e
5	object initialization join point	aspect a	advice b with pointcut c	join point d	class e	c picks out join point d associated with e
6	attribute reference join point	aspect a	advice b with pointcut c	join point d	class e, class f	c picks out join point d associated with e and f
7	attribute assignment join point	aspect a	advice b with pointcut c	join point d	class e, class f	c picks out join point d associated with e and f
8	handler execution join point	aspect a	advice b with pointcut c	join point d	class e	c picks out join point d associated with e
9	advice execution join point	aspect a	advice b with pointcut c	join point d	advice e	c picks out join point d associated with e

Figure 11 Dépendances engendrées par les joinpoint

La figure 12 [Bar 06] résume les différents types de dépendances basées sur l'utilisation des identificateurs.

#	Mechanism	Client Class	Client Item	Server Item	Server Class	Description
1	attribute definition	aspect a	attribute b, attribute intertype c	class d	class d	d is the type of b or c
2	local variable	aspect a	method b, advice c, method intertype d	class e	class e	e is the type of a local variable of b, c or d
3	return type	aspect a	method b, advice c, method intertype d	class e	class e	e is the return type of b, c or d
4	parameter	aspect a	method b, advice c, method intertype d, pointcut e	class f	class f	f is the type of a parameter of b, c, d or e
5	throws clause	aspect a	method b, advice c, method intertype d	class e	class e	e appears as a type in a throws clause of b, c or d
6	cast expression	aspect a	method b, advice c, method intertype d, pointcut e	class f	class f	c is the type in a cast expression in b, c, d or e
7	inheritance	aspect a	aspect a	aspect b ≠ a, class c, interface d	aspect b ≠ a, class c, interface d	a is directly derived from b, c or d
8	attribute intertype declaration	aspect a	attribute b	class c	class c	b is defined for c
9	method intertype declaration	aspect a	method b	class c	class c	b is defined for c
10	declare precedence	aspect a	aspect a	aspect b	aspect b	b appears as a type in a declare precedence statement
11	declare parents	aspect a	aspect a	class b, interface c	class b, interface c	b or c appear as a type in a declare parents statement
12	declare soft	aspect a	aspect a	class b derived from Exception	class b derived from Exception	b appears in a declare soft statement

Figure 12 Dépendances engendrées par les identificateurs

On retrouve aussi une autre forme de dépendances qui se base sur les invocations (figure 13 [Bar 06]).

#	Mechanism	Client Class	Client Item	Server Item	Server Class	Description
1	invocation	aspect a	method b, advice c, method intertype d, pointcut e	method f, method intertype g	class h	b, c, d or e invokes f or g
2	invocation parameter	aspect a	method b, advice c, intertype method d, pointcut e	class f	class f	f is the type of a parameter of a method invoked by b, c, d or e

Figure 13 Dépendances engendrées par les invocations

La granularité nous donne le niveau de détail pour les différents types de couplage. Le framework proposé par Briand et al. [Bri 99] comprend deux facteurs : le domaine où la mesure est effectuée et comment la connexion est évaluée. Le domaine de mesure nous indique quelles composantes sont calculées. En AspectJ, le domaine de mesure fait référence au : join point, attribute, method, pointcut, intertype declaration, advice, aspect, class, interface et le système. Briand et al. [Bri 99] suggèrent 6 types de mesures de connexions. On peut voir le résultat à la figure 14 (niveau membre) et la figure 15 (niveau aspect/classe).

#	Type	Example
1	count individual connections	for each advice the number of references to attributes
2	count the number of distinct members at the other end of the connection	for each advice the number of attributes referenced

Figure 14 Mesure de connexion pour le niveau membre *

#	Type	Example
1	add up the connections counted as in Table 7 #1 for each member of the entity	the total number of attribute references by advice in the aspects
2	add up the numbers of connections counted as in Table 7 #2 for each member of the entity	add up the number of attributes referenced by each advice of the aspect
3	count the number of distinct members at the end of the connections starting from or ending in members of the class	the number of attributes referenced by advice of the class
4	for an entity count the number of other entities to which there is at least one connection	the number of classes which have an attribute that is referenced by an advice of the aspect

Figure 15 Mesure de connexion pour les aspects/classes

3.3 Ordonnancement

L'ordonnancement représente l'ordre dans lequel vont être intégrées les différents composants de notre système. Il est intimement lié au degré de dépendances entre les classes et/ou les aspects. Ces dépendances (différents types), seront utilisées au niveau de notre stratégie en tant que critère lors du processus d'intégration. Si on se réfère aux figures 3 et 4 précédentes, nous sommes en présence de cycles de dépendance autant pour les classes que pour les aspects.

Il faudra donc établir une stratégie permettant d'assurer une intégration complète et optimale tenant compte des liens qui existent entre les classes et les aspects (différents niveaux : classes-classes et classes-aspects). Avant d'entrer dans

la partie consacrée à la stratégie que nous proposons, nous allons introduire dans ce qui suit quelques éléments clés utilisés au niveau de la stratégie que nous préconisons.

3.4 Notion de bouchon de test

Le bouchon de test peut représenter un composant non encore intégré dont on simule le comportement afin d'intégrer un autre composant qui l'utilise. En faisant référence à la figure 3, le but ultime du bouchon de test est primordial puisqu'il permet de briser certains liens de dépendances qui autrement rendent notre système cyclique. Prenons, par exemple, le choix de la classe C dans la figure 3 précédente comme bouchon de test. L'ordonnancement serait l'intégration de la classe B puis l'intégration de la classe A et enfin la classe C en dernier lieu. Nous pouvons alors briser le lien [B C] ce qui nous donne un petit système acyclique représenté par la figure 16:

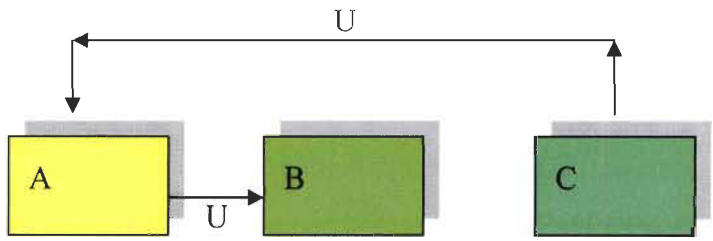


Figure 16 Création d'un bouchon de test (1)

La création de bouchon de test est essentielle lorsque nous sommes en présence de cycles de dépendances. Par contre, en l'absence de cycle de dépendances, le bouchon de test n'a aucun rôle à jouer [Bad 04]. Sa création est inutile.

3.4.1 Bouchon réel

Le bouchon réel permet la simulation du comportement entier du composant. Dans la figure 16, la classe C représente un bouchon réel. Ce bouchon est donc utilisé si on veut intégrer toutes les classes qui dépendent de la classe dont il simule le comportement.

3.4.2 Bouchon spécifique

Le bouchon spécifique permet la simulation d'un seul service du composant utilisé. Soit la figure 17 :

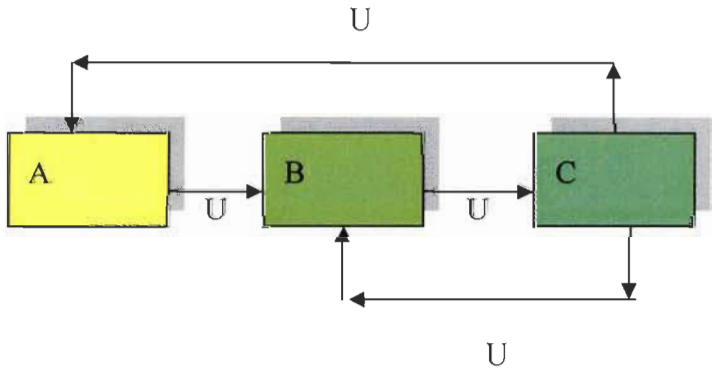


Figure 17 Création de deux bouchons spécifiques

Dans le cas qui nous concerne, nous pouvons identifier deux bouchons spécifiques pour la classe B. Ce sont les bouchons [C B] et [A B]. Le bouchon spécifique [C B] sera utilisé uniquement lors de l'intégration de la classe C, tandis que le bouchon spécifique [A B] servira à l'intégration de la classe A. On devra aussi départager nos choix lorsqu'il s'agira de tenir compte de la complexité des bouchons de test. Le choix reviendra au testeur.

3.5 Minimisation de bouchons de test

La minimisation des bouchons de test dans le cadre d'une stratégie d'intégration donnée est étroitement liée à la manière de détecter et de briser les cycles de dépendances pour obtenir un système dit « acyclique ». On notera qu'il peut y avoir plusieurs bouchons de test pour un système donné, en fonction des choix effectués dans la démarche de l'intégration. Le plus souvent, ces choix sont éclairés par des stratégies d'intégration. Cependant, peu importe la démarche engagée, il faut garder à l'esprit que la réduction du nombre de bouchons de test aura un impact direct sur l'effort de test global permettant ainsi d'optimiser une bonne partie du travail.

3.6 Stratégies d'intégration

3.6.1 Paradigme orienté objet

Plusieurs stratégies ont été proposées dans la littérature pour résoudre les divers problèmes liés aux tests d'intégration des classes dans les systèmes orientés objet [Tai 99, Bri 02, Tra 00, Bad 05, Blé 05, etc.].

Harrold et al. [Har 92] proposent une stratégie dite incrémentale en vue de tester les classes qui sont liées entre elles. Tout d'abord, les classes de base sont testées en tenant compte des spécifications et des cas de test (test cases), puis les sous-classes sont testées de façon incrémentale permettant ainsi de réduire l'effort de test. Soundarajan et al. [Sou 01] reprennent la même stratégie en utilisant le principe de l'héritage qui sera utilisé pour tester le système de façon graduelle. Ils proposent une technique basée sur les spécifications et les tests des classes dérivées à partir des classes de base.

La stratégie proposée par Tai et Daniels [Tai 99] se résume en deux volets. En premier lieu, il s'agit d'assigner des nombres de niveau majeur en se référant aux relations d'agrégation et d'héritage. Puis, à chaque niveau majeur déterminé, il s'agit d'assigner aux classes des nombres de niveau mineur en se basant sur les relations d'association. Ces nombres de niveau majeur et mineur permettent de définir l'ordre d'intégration des classes. Ainsi, les classes auront un couple niveau majeur(I) et mineur(J) qu'on dénommera couple (I, J). Afin de procéder à l'élimination des cycles de dépendances, les auteurs utilisent une fonction de pondération. Cette fonction est représentée par un poids (P) qui correspond à la somme des liens entrants (nœud de départ) et des liens sortants (nœud d'arrivée). Enfin, pour éliminer les cycles de dépendances, la relation ayant le poids le plus élevé sera retirée du modèle.

L'approche de Le Traon et al. [Tra 00] se sert de l'algorithme de Tarjan [Tar 72] afin de trouver les composants fortement couplés (SCC). Pour éliminer les cycles de dépendances, les auteurs définissent un nouveau type de dépendance appelé *frond* désignant un lien allant de la classe B à la classe A où A est l'origine du cycle. Ils se basent sur une fonction de pondération qui permet d'attribuer un certain poids pour chacune des classes du SCC concerné. Celui-ci est défini comme étant la somme des *frond* entrants et sortants d'une classe. Ainsi, la classe ayant le poids le plus élevé est retirée du modèle et on poursuit la démarche récursivement.

La stratégie de Briand et al. [Bri 02] utilise une approche similaire à celle de Le Traon. À partir de l'intérieur de chaque SCC c'est-à-dire pour toute composante fortement couplée, il s'agit alors de calculer le poids pour chaque relation d'association. La fonction de pondération se définit comme étant la somme des liens entrants pour la classe A multipliée par la somme des liens sortants de la classe B. Ainsi, la relation d'association ayant le poids le plus élevé sera retirée au sein du SCC.

Les travaux de Badri et al. [Bad 05] proposent une nouvelle stratégie

d'intégration qui tient compte des interactions entre les classes (collaborations multiples). Leur stratégie, nommée B3, se base sur le modèle MDC qui se veut un modèle de dépendances entre les classes et qui découle du diagramme de classes et des diagrammes de collaboration entre classes UML. Dans leur modèle, les relations d'association et d'agrégation sont remplacées par la relation d'utilisation. La relation d'héritage est maintenue. L'intérêt de cette approche pour les tests d'intégration des classes se situe au niveau de la précision du diagramme de base et de la méthodologie appliquée. La minimisation de bouchons de test amène les auteurs à identifier deux types de dépendances : cycle effectif et cycle non-effectif. Les auteurs comparent leur approche à celles proposées par d'autres auteurs dans le domaine. Les résultats obtenus grâce à cette stratégie sont meilleurs que ceux obtenus par les autres stratégies; la complexité et le nombre des bouchons de test sont réduits.

3.6.2 Paradigme orienté aspect

Dans le domaine aspect, l'intérêt vis-à-vis de la problématique des tests d'intégration est récent. En effet, très peu de travaux ont été réalisés dans ce contexte. S'inspirant de la stratégie de Briand et al. [Bri 03], Ré et al. [Reg 07] proposent une stratégie permettant l'ordonnancement des classes et des aspects dans un système orienté aspect afin de minimiser le nombre de stubs (bouchons) lors des tests d'intégration. Cet ordonnancement est basé sur un modèle de type de dépendances entre classes et aspect conçu à partir de considérations syntaxiques et sémantiques d'AspectJ. Dans leur stratégie, les classes et les aspects sont traités de la même façon : c'est-à-dire qu'ils procèdent à l'ordonnancement des classes et des aspects en même temps; testent aussi bien une classe qu'un aspect et prévoient pour les tester soient des stubs de type classes ou bien de type aspects. Enfin, ils soutiennent que leur stratégie est généraliste et peut être appliquée à plusieurs langages. Il ne semble pas définir de différences marquantes entre le test des classes et des aspects. Cependant, il existe des différences fondamentales entre les deux types d'entités. Les classes sont des entités complètes en elles-mêmes au niveau de leurs fonctionnalités primaires et peuvent donc être testées

seules. Mais comme l'article [Ale 04] l'a bien souligné, les aspects n'ont pas d'existence propre en dehors de leurs contextes d'intégration. C'est donc à travers leur influence dans les différents contextes qu'ils prennent une forme tangible.

D'autres travaux dans le domaine aspect ont porté sur le test, d'une manière générale, ou bien sur le test unitaire au niveau des applications orientées aspect. Parmi ces différents travaux, certains abordent les caractéristiques des aspects, d'autres présentent les bases sur lesquelles le test aspect devrait se fonder et d'autres encore proposent différentes méthodes de test orienté aspect. Parmi ces travaux, nous pouvons citer [Xu 06b, Mah 04, Vid 01, Anb 01]. Ce dernier, en particulier, s'est intéressé à la génération de code permettant la gestion des préoccupations transversales à l'aide de diagrammes d'états.

Zhao Zhao 01], [Zha 02], [Zha 03] propose une méthode de tests unitaires pour les classes dans un système orienté-aspect. Sa méthodologie est basée sur des tests issus de diagramme de flot de contrôle. La méthode préconisée par Zhao se situe à trois niveaux soit intra-modulaire (tests basés sur les méthodes, advices, etc), inter-modulaires mesurant les interactions entre les modules puis intra-classe/intra-aspects où l'on retrouve le test pour une classe ou un aspect. Le même auteur [Zha 04] utilise une technique de sélection de tests afin de produire des tests de régression basé sur le flot de contrôle pour des systèmes orientés-aspects.

Xu [Xu 01] préconise une approche basée sur les diagrammes d'états. Les séquences de test produites permettent de vérifier l'ensemble des possibilités du diagramme. Cette technique semble prometteuse mais se limite à tester l'ensemble classe/aspect de façon modulaire ce qui veut dire que s'il y a ajout ou modification d'un aspect, il faut retester le tout. Dans [Xu 02], il propose une méthode de test unitaire basé sur le flot de contrôle. Cette approche permet de fusionner les diagrammes d'états des classes ainsi que ceux des aspects appelé Scope State Model (ASSM).

CHAPITRE 4

PRÉSENTATION DE L'ALGORITHME B3

4.1 Étapes fondamentales

Ce chapitre présente la stratégie (Algorithme B3) proposée par Badri et al. [Bad 05]; stratégie de test d'intégration pour les systèmes orientés objet. La stratégie de test d'intégration pour les systèmes orientés aspect, que nous proposons, correspond à une extension de celle-ci. Le modèle de dépendances entre classes MDC, sur lequel s'appuie la stratégie proposée par Badri et al. [Bad 05], tient compte des interactions qui existent entre les différentes classes d'un système. Il dérive du diagramme de classes UML auquel sont intégrées les différentes interactions entre les classes décrites dans les diagrammes de collaboration UML. Les différentes relations d'association et d'agrégation présentes dans le modèle sont remplacées par la relation d'utilisation. Le modèle de dépendance utilisé contient donc deux types de relations de base: la relation d'héritage et la relation d'utilisation. Les étapes fondamentales de cette stratégie sont :

4.2 Affectation d'un nombre niveau majeur

Le but de cette étape est de scinder le Modèle (MDC) en plusieurs niveaux majeurs. Le critère de séparation utilisé est la relation d'héritage. Cette démarche permet de faire une première élimination de cycle de dépendance.

4.3 Détermination et élimination des cycles de dépendances

Il s'agit de détecter et d'éliminer les cycles de dépendances présents dans le

modèle. En premier lieu, les auteurs s'intéressent aux cycles de dépendances à l'intérieur de chaque niveau majeur. Ils évaluent pour chaque classe le poids généré pour les liens sortants qui ont contribué au cycle de dépendances; la classe qui a le poids le plus élevé sera considérée comme bouchon de test pour sortir des cycles de dépendances dans lesquels elle est impliquée. Le poids, pour une classe donnée, est défini comme étant le nombre total de ses liens sortants impliqués dans des cycles de dépendances. Suite au choix du bouchon de test, tous les liens entrants (vers cette classe) seront retirés. Ce processus est réitéré jusqu'à ce qu'il n'y est plus de cycles de dépendances au niveau majeur. Quelques cas peuvent se présenter, le traitement de parité y apportera quelques nuances.

4.3.1 Traitement de parité

Plusieurs cas possibles peuvent se présenter :

1) Si l'une des classes de niveau N utilise une classe de niveau N+1, celle-ci sera utilisée comme bouchon de test. La classe appelante est toujours intégrée après la classe appelée.

2) Si pour une classe donnée, elle est impliquée dans un cycle de dépendances qui n'implique aucune autre classe de même parité, celle-ci sera utilisée comme bouchon de test.

3) Si plusieurs classes de parité égale sont impliquées dans un cycle de dépendances inter niveau, la fonction de pondération, définie précédemment pour sortir du cycle, sera utilisée.

4) Si la parité persiste, les auteurs utilisent les détails issus du modèle initial suite aux interactions parmi les classes. Ainsi, ils ont une bonne information quant à la

complexité des interactions (cycles) pouvant survenir entre les classes et choisir le bouchon le plus complexe.

4.4 Affectation d'un nombre niveau mineur

Une fois que tous les cycles à chacun des niveaux majeurs sont brisés, les auteurs procèdent à l'affectation du nombre niveau mineur pour chacune des classes du niveau majeur. A chaque classe du niveau majeur, on attribue une valeur déterminée en fonction des relations de dépendances qu'elle peut avoir avec les autres classes du système. Ce calcul est effectué d'abord, au sein de chaque niveau majeur sans tenir compte des relations inter-niveau, on parle dans ce cas de nombre niveau mineur provisoire. Puis, à partir des relations inter-niveaux, un nombre niveau mineur définitif est affecté.

4.5 Ordonnancement des classes

Cette étape permet l'ordonnancement des composantes dans le système. Il se base sur les couples (niveau majeur, niveau mineur) qui ont été préalablement déterminés auparavant. Les classes dont le niveau mineur est le plus faible seront intégrées avant celles dont le niveau est le plus élevé. En cas de parité au niveau mineur, le niveau majeur aura priorité.

CHAPITRE 5

EXTENSION DE L'ALGORITHME B3

5.1 Présentation

La nouvelle stratégie que nous proposons est une extension de la stratégie qui a été développée pour les applications orientées objets [Bad 05]. Il s'agit d'une stratégie basée sur le Modèle de Dépendances entre Classes (MDC) qui dérive du diagramme de classes de conception UML auquel ont été intégrées les interactions entre classes décrites dans les diagrammes de collaboration entre classes. Les différentes relations d'association et d'agrégation présentes dans le modèle sont remplacées par les interactions entre classes qui les supportent. Le modèle de dépendance de la stratégie contient donc deux types de relations de base : la relation d'héritage et la relation d'utilisation. Une classe A utilise une classe B si au moins une des méthodes de la classe A appelle au moins une des méthodes de B. La figure 18 présente un exemple du modèle MDC, utilisé par la nouvelle stratégie, auquel seront greffés différents aspects.

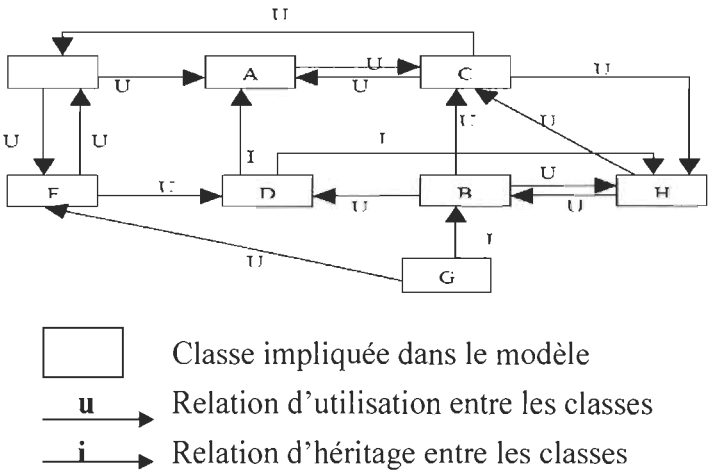


Figure 18 Modèle de dépendances entre les classes.

Nous avons choisi la stratégie basée sur l'algorithme B3 car elle présente de bons résultats comparée à d'autres dans le domaine [Bad 05]. En effet, cette stratégie a été comparée à trois stratégies représentatives des stratégies existantes dans le contexte objet (stratégie de Taï et Daniels [Tai 99], stratégie de Briand et al. [Bri 02] et stratégie de Triskell [Tri 02]).

L'originalité de l'approche que nous préconisons pour le test d'intégration des classes et des aspects se situe, d'une part, au niveau de la précision du diagramme de base et, d'autre part, dans la démarche même de la stratégie. L'apport se situe surtout au niveau de la minimisation du nombre de bouchons de test et de l'effort de leur conception. La minimisation des bouchons de test est due aussi bien à la précision du diagramme de base utilisé par la stratégie, qu'à la démarche même de la stratégie. La précision du diagramme nous permet, à travers la démarche d'intégration, d'identifier deux types de cycle de dépendances. Les cycles de dépendance effectifs et les cycles de dépendance non effectifs. Les figures suivantes illustrent un exemple de chaque type de cycle. Le premier est un cycle de dépendances effectif (Figure 19) et le second est un cycle de dépendances non effectif (Figure 20). Par ailleurs, la connaissance de la nature des interactions (entre les classes, les classes et les aspects, et les aspects) ainsi que de leur complexité est pertinente. Elle nous permet, d'une part, en cas de parité entre classes, de faire un choix éclairé quant à la classe à utiliser comme bouchon de test. D'autre part, en cas de plusieurs aspects associés à une classe donnée, de choisir l'aspect le plus complexe et qui n'a aucune autre dépendance avec d'autres aspects (processus d'intégration du plus complexe vers le moins complexe). Ces différents points seront discutés en détail dans les prochaines sections.

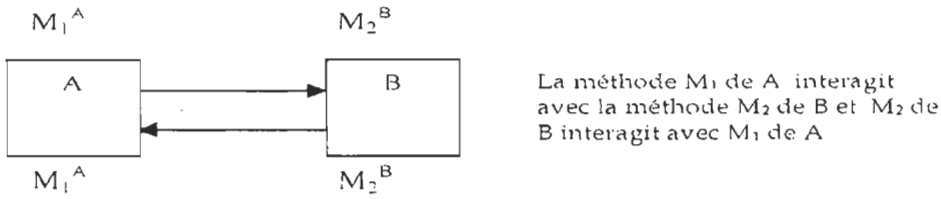


Figure 19 Présence de cycle de dépendance effectif

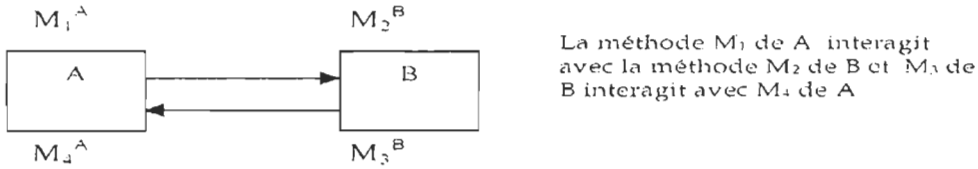


Figure 20 Présence de cycle de dépendance non effectif

La notion de cycle de dépendances non effectif a conduit à définir le principe de l'intégration partielle. La figure 21 illustre l'intégration en présence d'un cycle de dépendances non effectif. À travers cette illustration nous présentons la notion de l'intégration partielle; intégration d'une classe sans une ou plusieurs de ses méthodes. La (ou les) méthode(s) non intégrées sont celles qui nous permettent de sortir du ou des cycles de dépendance, non effectif. La figure 22, présente l'intégration, dans le cas d'un cycle de dépendances effectif.

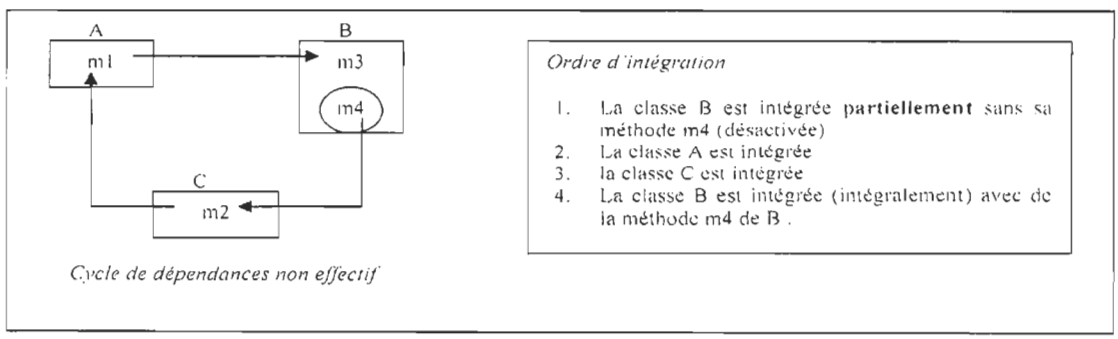
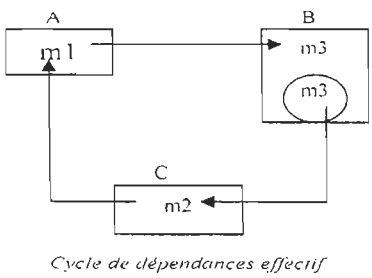


Figure 21 Intégration d'un cycle de dépendances non effectif



Cycle de dépendances effectif

Ordre d'intégration

Dans ce cas, nous sommes en présence d'un cycle effectif. Il nous faut donc absolument un bouchon de test. Si nous gardons le même ordre d'intégration que dans la figure 12, alors nous ferons de la classe C un bouchon de test.

1. La classe B est intégrée en utilisant un bouchon de la classe C
2. La classe A est intégrée
3. la classe C est intégrée

Figure 22 Intégration d'un cycle de dépendances effectif

5.2 Méthodes d'intégration : Principales étapes

La méthode d'intégration que nous proposons est organisée en cinq principales étapes:

- Affectation d'un nombre niveau majeur,
- Détermination et élimination des cycles de dépendances (effectifs et non effectifs),
- Affectation d'un nombre niveau mineur,
- Ordonnancement des classes,
- Intégration des classes et des aspects.

5.2.1 Affectation d'un nombre niveau majeur

Le but de cette étape consiste à scinder le diagramme original en plusieurs niveaux. L'héritage (H) servira de critère de séparation. Cette démarche permet d'isoler les relations d'héritage lors d'une première suppression des cycles de dépendance. La figure 23 illustre l'affectation des nombres niveau majeur au modèle.

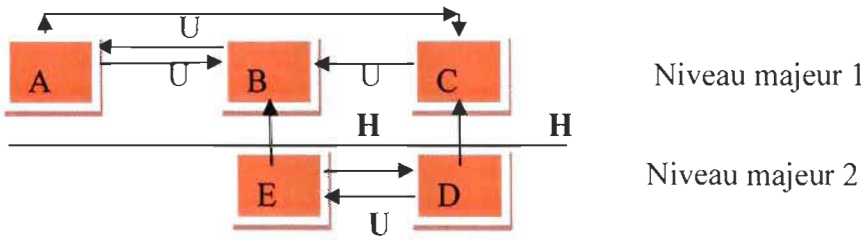


Figure 23 Affectation du nombre niveau majeur en fonction de l'héritage

5.2.2 Détermination et élimination des cycles de dépendances

Cette étape consiste à détecter et à éliminer les cycles de dépendances présents dans le modèle. Nous nous attaquons, dans un premier temps, aux cycles de dépendances à l'intérieur de chaque niveau majeur. Pour ce faire, nous calculons le poids de chacune des classes du niveau majeur considéré. Le poids des classes nous permet de déterminer les bouchons de test. Le poids de chaque classe est évalué en fonction de ses liens sortants impliqués dans le cycle: *poids de la classe C = somme de ses liens sortants impliqués dans un ou des cycles de dépendances*. La classe ayant le poids le plus élevé est choisie comme bouchon de test. Après le choix du bouchon, tous ses liens entrants sont retirés du diagramme. Ensuite, nous calculons à nouveau le poids des classes et ainsi de suite jusqu'à ce qu'il n'y ait plus de cycles de dépendances au sein du *niveau majeur*. L'élimination des cycles de dépendances est basée sur un processus descendant. Il commence au niveau le plus élevé et descend jusqu'au niveau le plus bas.

Selon le niveau majeur 1 à la figure 23, la classe A est un bouchon de test puisqu'elle possède le poids le plus élevé selon ses liens sortants, on peut procéder alors à une première élimination de cycle de dépendance qui nous donnera la figure 24.

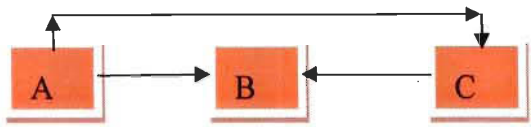


Figure 24 Élimination du cycle de dépendance au niveau majeur 1

Le lien B vers A est ainsi éliminé. Ce processus itératif se poursuit jusqu'à ce qu'il n'y ait plus de cycle de dépendances. Cependant, quelques cas peuvent se présenter, le traitement de parité apporte quelques nuances. Soit les cas suivants :

- Si l'une des classes du niveau N utilise une classe de niveau n+1, celle-ci sera utilisée comme bouchon de test afin d'éliminer le cycle de dépendances. La classe appelante est toujours intégrée après la classe appelée. Dans la figure 25 suivante, il y a parité entre les classes B et H de niveau 1. Vu que la classe B utilise la classe D qui est de niveau 2, la classe B sera considérée comme bouchon de test.

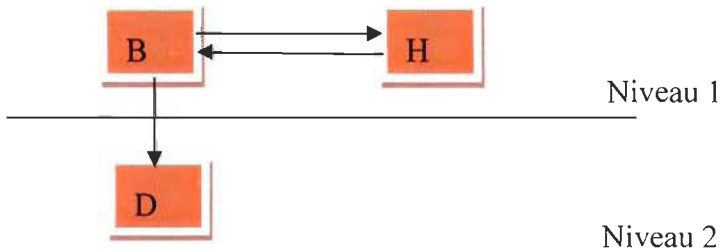


Figure 25 Parité entre deux classes

- Si une classe donnée est impliquée dans un cycle de dépendances qui n'implique aucune autre classe de même parité, celle-ci sera utilisée comme bouchon de test. Ceci permettra de sortir du cycle de dépendances du niveau N, d'une part, et du cycle inter-niveaux engendré par les relations de la classe avec les autres classes du modèle, d'autre part. Dans la figure 26, nous remarquons qu'il y a parité entre la classe B et A. Vu que A et B sont effectivement impliqués dans un SCC inter niveaux; c'est-à-dire en présence de liens entre composantes qui sont fortement couplées. Nous appliquerons la fonction

de pondération pour sortir du cycle de dépendances. On supposera que tous les liens sont de type utilisation.

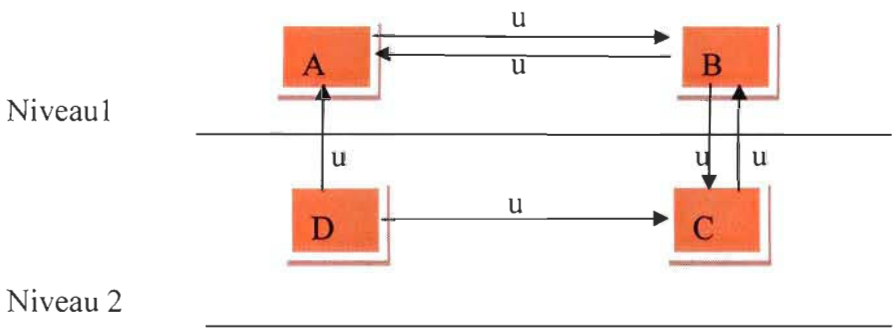


Figure 26 Parité entre deux classes, dont l'une est impliquée dans un cycle inter niveau

- Si plusieurs classes de parité égale sont impliquées dans un cycle inter niveaux, on applique la fonction de pondération préconisée précédemment pour sortir du cycle.

5.2.3 Affectation d'un nombre niveau mineur

Si une classe B dépend d'une classe A, alors la classe B se verra affectée le nombre *niveau mineur* de la classe A augmenté de 1. A ce stade du processus d'intégration, les valeurs attribuées aux classes sont provisoires. Elles sont calculées d'abord au sein de chaque niveau majeur sans tenir compte des dépendances inter niveaux. Ensuite, à partir des dits liens inter niveaux, nous affecterons un nombre niveau mineur définitif. En faisant référence à la figure 23, nous remarquons que les classes A, B, C, D sont de *niveau mineur provisoire* (P) égale à :

$P_a = 1$, $P_b = 2$, $P_d = 1$, $P_c = 1$ puis en tenant compte des relations inter *niveaux mineurs définitifs* nous aurons $D_a = 2$, $D_b = 3$, $D_d = 1$, $D_c = 4$. Les résultats sont exprimés à la figure 27.

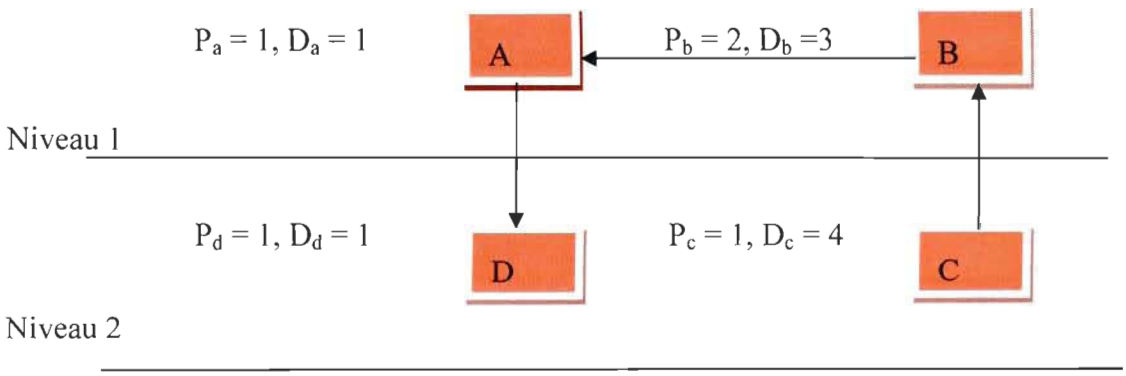


Figure 27 Affectation du nombre mineur aux classes

5.2.4 Ordonnancement des classes sans les aspects

Le processus d'ordonnancement consiste à déterminer l'ordre d'intégration des classes sans tenir compte dans un premier temps des aspects. Il est basé sur les couples (*niveau majeur*, *niveau mineur*) déterminés précédemment. Les classes dont le niveau mineur est plus faible sont intégrées avant celles dont le *niveau mineur* est élevé. En cas de parité du *niveau mineur*, le niveau majeur est utilisé pour permettre d'effectuer une sélection finale. Si on se rapporte à la figure 28, la classe B est intégrée avant la classe A, car le niveau mineur de B (1) est inférieur à celui de A (2).

Classe A : Niveau majeur = 1 Niveau mineur =2
Classe B : Niveau majeur = 2 Niveau mineur =1

Figure 28 Processus d'ordonnancement des classes

L'ordonnancement de l'intégration des classes pour le modèle initial, nous donnera :

- A est intégrée en utilisant le bouchon C
- H est intégrée en utilisant les bouchons B et C
- E est intégrée en utilisant le bouchon de F
- D est intégrée en utilisant les classes A et H
- C est intégrée en utilisant les classes A, E et H
- B est intégrée en utilisant les classes C, D et H
- F est intégrée en utilisant les classes D et E
- G est intégrée en utilisant les classes B et F

5.2.5 Intégration des classes et des aspects

Une fois l'ordre d'intégration des classes établi, on procède à l'intégration des classes et des aspects. D'une manière générale, il s'agit de greffer à chacune des classes obtenues l'ensemble des aspects qui lui sont reliés. Plusieurs scénarios peuvent se présenter dans ce contexte: un seul aspect est relié à la classe (figure 30), un aspect est relié à plusieurs classes (figure 31) ou bien plusieurs aspects sont reliés à la classe (figure 32). Le processus d'intégration des aspects est basé sur le modèle des types de dépendances (chapitre 3 section3.2) pouvant exister entre les classes et les aspects établis par [Reg 07] à partir des constructions syntaxiques et sémantiques d'AspectJ (figure 29).

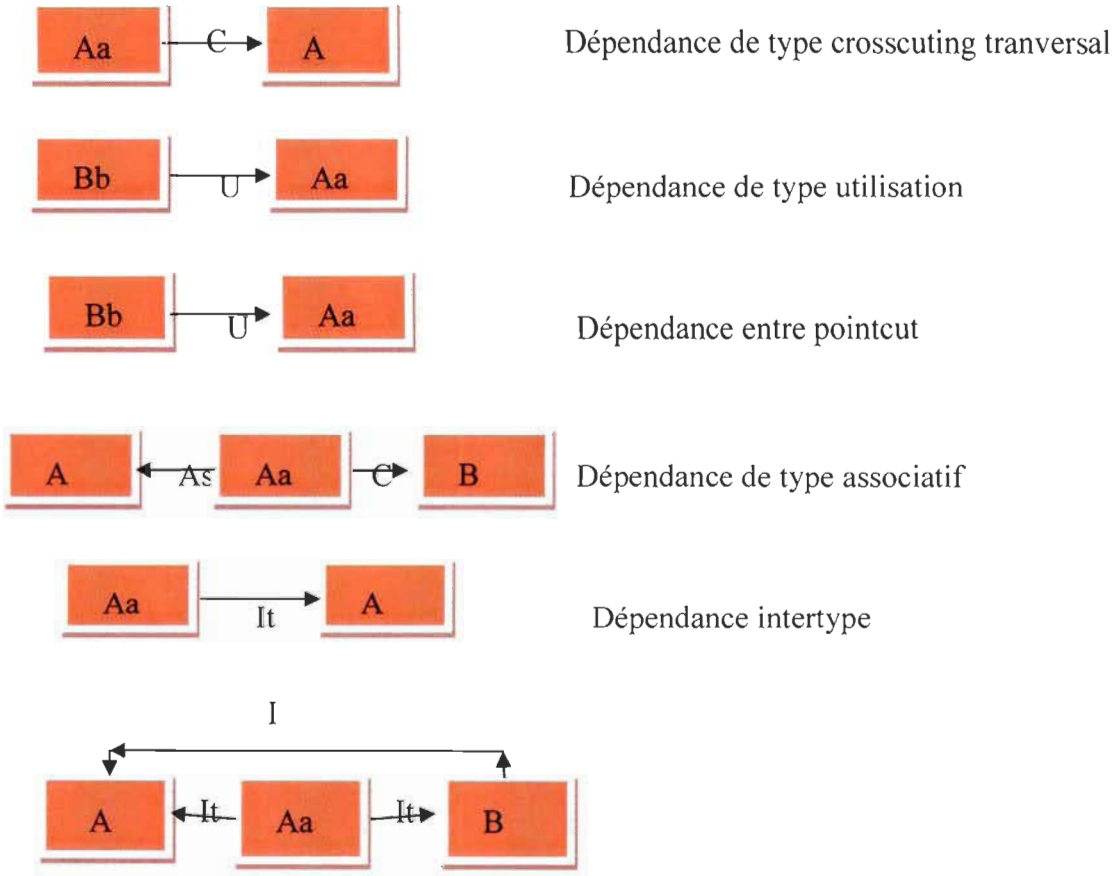


Figure 29 Types de dépendances entre aspects et classes [Reg 07]

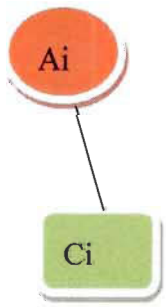


Figure 30 Aspect lié à une seule classe

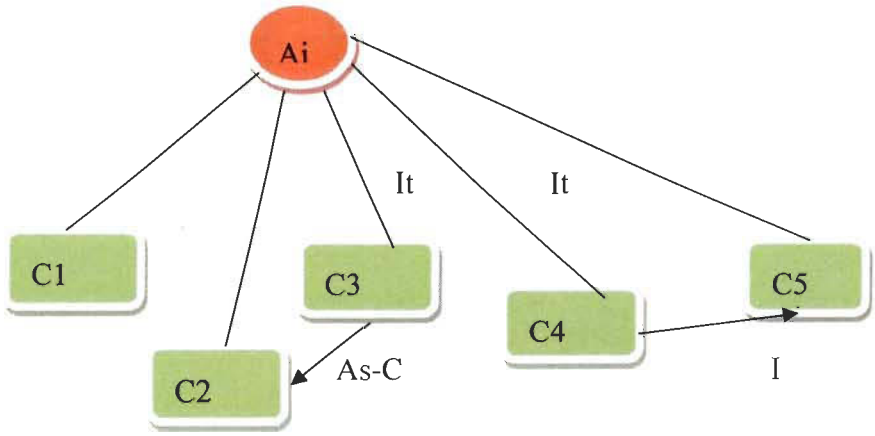


Figure 31 Aspect lié à plusieurs classes

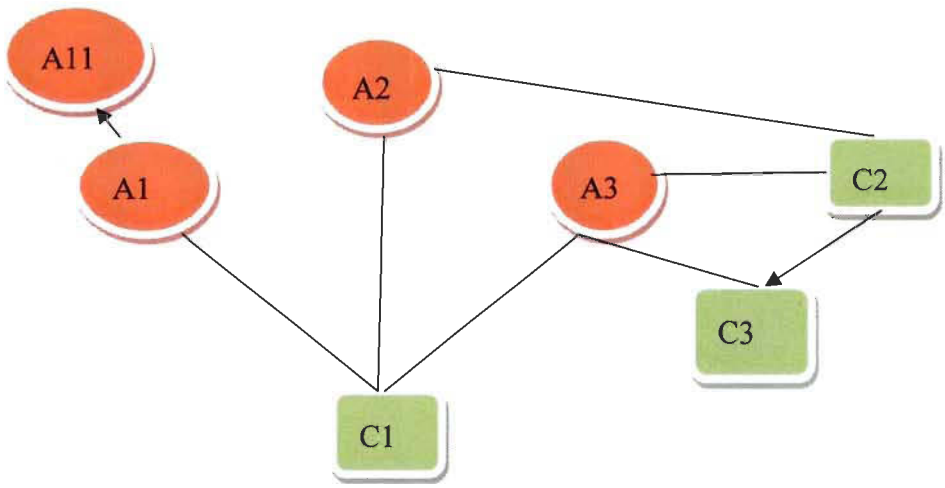


Figure 32 Classes liées à plusieurs aspects

Il s'agit, en fait, de faire un zoom sur la classe en question et voir de façon détaillée comment les aspects se greffent à la classe. Dans ce contexte, le processus d'intégration des aspects à la classe se fait de façon incrémentale en tenant compte du type de dépendance, de la complexité, et du couplage.

Le processus d'intégration de la classe se fait en deux étapes :

- on procède d'abord au test de la classe conformément à l'approche de test présenté par [Bou 07]. Il s'agit d'une technique de test unitaire orientée aspect basée sur le comportement dynamique décrit par le diagramme d'états de la classe. Le but ultime de cette démarche consiste d'abord à s'assurer que le comportement de la classe n'est pas altéré par l'ajout des aspects. Plusieurs critères de test ont été introduits afin de déterminer ce que les tests doivent couvrir. Par ailleurs, cette approche est supportée par un outil de génération et d'exécution de tests intégrant les critères de test développés.

- Par la suite on procède à l'intégration de la classe avec les aspects au reste des classes déjà intégrées en définissant, s'il y a lieu, soient des bouchons réels ou bien des bouchons spécifiques. L'intégration des d'aspect se fait de façon incrémentale.

Pour mener à bien le processus des tests d'intégrations des classes et des aspects, on utilise diverses informations :

- L'ordonnancement de l'intégration des classes obtenu à partir de B3,
- Une liste LAi rattachée à chaque aspect dans laquelle on spécifie sa complexité, l'ensemble des classes auxquelles est rattaché l'aspect Ai, le type de dépendance et le couplage avec chaque classe, et pour chaque classe reliée à l'aspect l'ensemble des classes qui lui sont reliées par le biais de l'aspect en question.
- Une liste CI des classes déjà intégrées, au départ cette liste est vide.

Les principales étapes relatives à l'intégration des aspects sont :

Démarche :

```
Début                                   /* début du bloc 1*/
Tant qu'il y a des classes à intégrer faire
Pour chaque classe Ci obtenue selon l'ordonnancement à l'étape précédente faire:
Si un seul aspect Ai se greffe à cette classe
Alors
        Si  Ai est lié uniquement à la classe Ci
        Alors
        intégrer Ai à la classe Ci . procéder au test de Ci et m.a.j CI
        Sinon  / *Ai  est lié à plus d'une classe*/
        Debut                                   /* début du bloc 2*/

                Si les classes dont dépend l'aspect sont intégrées
                Alors    intégrer Ai à la classe Ci . procéder au test de Ci et m.a.j  CI
                Sinon
                                définir des classes vides pour les classes ayant des dépendances de
                                types It  et I (**)
                                procéder au test de la classe avec l'aspect et m.a.j CI
                Fin si
        Fin                             /* fin du bloc 2*/
Sinon  /* plusieurs aspects qui se greffent à la classe*/
Debut                                   /* début du bloc 3*/
        classer les aspects par ordre de complexité et de couplage
       Tant que il y'a des aspects liés à la classe Ci
        Faire           /* intégration incrémentale*/

        Si l'aspect n'est lié à aucun autre aspect ni à aucune autre classes
        Alors       /* on commence par intégrer à la classe les aspects un à la fois */
                intégrer Ai à la classe Ci . procéder au test de Ci et m.a.j CI

        Sinon
              Si l'aspect est lié à d'autres classes
              Alors  faire la même chose que dans le bloc (2)
              Sinon Si l'aspect est lié à d'autres aspects
                        Alors  intégrer les aspects à la classe et procéder au test de la classe m.a.j CI
        FSi
        FTQ
Fin                                     /* fin du bloc 3*/
Finpour
FTQ
Fin                                     /* fin du bloc 1*/
```

Par ailleurs, dans le cas ou plusieurs aspects sont liés à une classe, on vérifie également s'ils se greffent autour du même point de jointure. Si c'est le cas, on procède au test de la classe en utilisant différentes combinaisons d'intégration des aspects pour

s'assurer que cela ne pose pas de problème.

Les règles d'intégration sont définies dans l'aspect à l'insu de la classe. L'aspect décrit, à l'aide de diverses constructions, comment cette intégration sera effectuée. Il est donc possible de procéder à l'intégration et au test de la classe sans que cela affecte le comportement de la classe. L'intégration se fait soit de façon dynamique ou bien de façon statique. Lorsque un aspect est lié à plusieurs classes, dans le cas de l'intégration dynamique il est possible de procéder à son intégration même si toutes les classes auxquelles il est lié ne sont pas intégrées.

Dans le cas de l'intégration statique (déclaration intertype), on définit des classes vides représentant les classes auxquelles l'aspect est lié de façon statique si celles–ci n'ont pas été intégrées et on procède à l'intégration de l'aspect avec la classe en question.

On n'aura pas à définir des classes vides, si durant le processus d'intégration, on définit des bouchons spécifiques ou des bouchons réels pour ces mêmes classes ; classes auxquelles est lié l'aspect.

Telle que définie, la stratégie d'intégration que nous avons développée pour les systèmes orientés aspect permet de préserver le nombre de bouchons de test (réel ou spécifique) obtenu par B3. Autrement dit, l'intégration des aspects aux différentes classes n'augmente pas le nombre de bouchons. L'effort, relatif à la définition de bouchons, restera le même avec ou sans l'intégration des aspects.

CHAPITRE 6

ÉTUDE DE CAS

6.1 Présentation de l'étude de cas

Nous présentons, dans ce chapitre, une étude de cas portant sur un exemple comportant plusieurs classes et aspects (figure 33). L'objectif, à travers cette étude, est de démontrer la faisabilité de l'approche que nous proposons pour le test d'intégration des applications orientées aspect. Par la suite, après cette première évaluation, on pourra l'implémenter et l'expérimenter sur des applications réelles (travaux futurs). L'expérimentation sera une opération relativement facile, sachant qu'une bonne partie des outils nécessaires au support de l'approche existe déjà (outils développés dans le cadre de travaux connexes à ce projet).

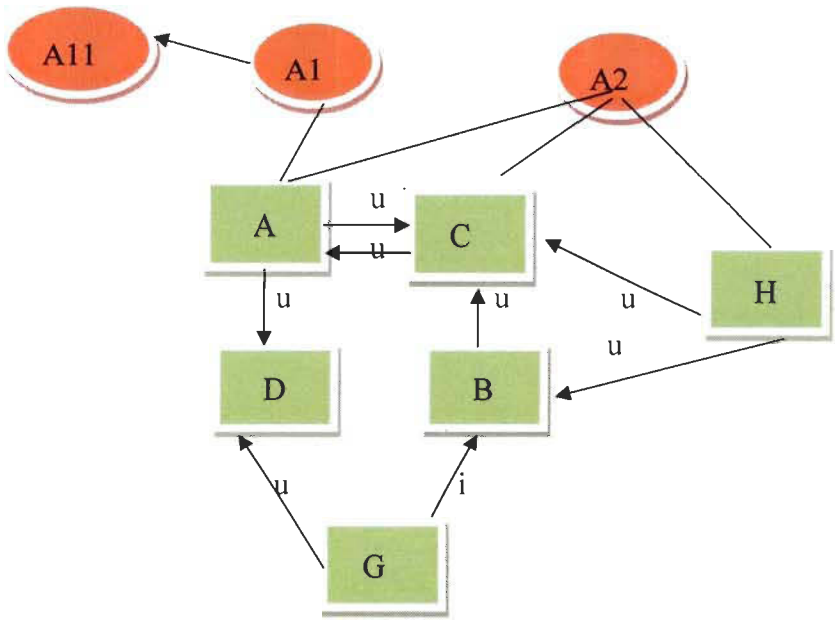


Figure 33 Étude de cas

6.2 Application de l'approche

Affectation d'un nombre niveau majeur

Le but de cette étape consiste à scinder le diagramme original en plusieurs niveaux. L'héritage (i) servira de critère de séparation. Selon L'approche, les classes A, B, C, D, H qui n'héritent d'aucune classe sont dans le niveau 1 et la classe G est dans le niveau 2 (figure 34).

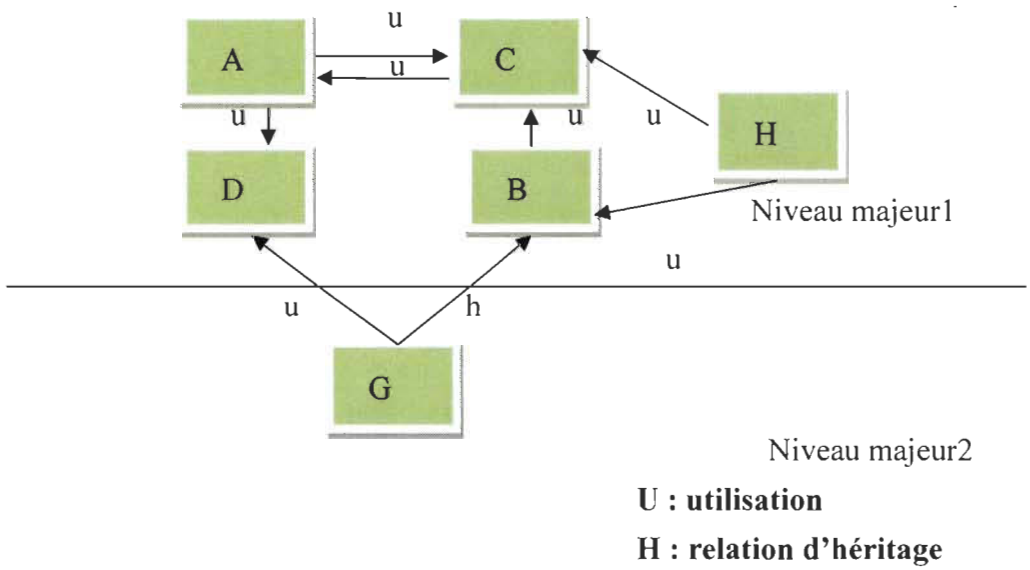


Figure 34 Affectation du niveau majeur en fonction de l'héritage

Élimination des cycles de dépendances au sein de chaque niveau majeur

A cet effet, nous calculons le poids de chacune des classes du niveau majeur considéré. Le poids des classes nous permet de déterminer les bouchons de test. Le poids de chaque classe est évalué en fonction de ses liens sortants impliqués dans le cycle: *poids de la classe C = somme de ses liens sortants impliqués dans un ou des cycles de dépendances*. A noter que les classes A, B. C, et H qui n'héritent d'aucune autre classe

seront de niveau majeur 1 tandis que la classe G qui hérite de B sera de niveau majeur 2. La classe ayant le poids le plus élevé est choisie comme bouchon de test.

Dans le cas présent, les classes A ou H possède le plus de liens sortants. Donc, elles seront utilisées comme bouchons de test. Débutons par la classe A, alors on peut procéder à une première élimination de cycle de dépendances pour les classes A, C, D et H qui nous donnera la figure 35.

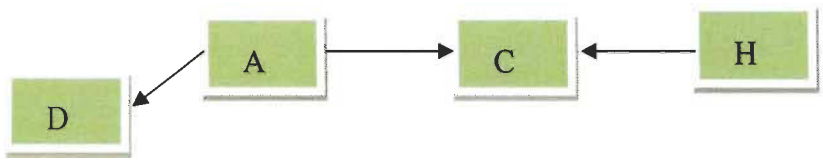


Figure 35 Élimination du cycle de dépendance au niveau majeur 1

Concernant les classes B et H, cette dernière possède 2 liens sortants par rapport à la classe C. La classe H sert de bouchon de test. Nous pouvons procéder à l'élimination d'un lien H et B. Pour le niveau majeur 1 nous obtenons à la figure 36:

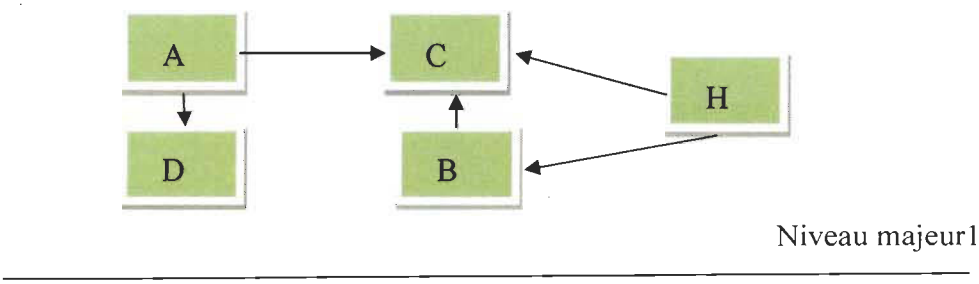


Figure 36 Élimination du cycle de dépendances au niveau majeur 2

Affectation d'un nombre niveau mineur

Lorsque tous les cycles de dépendances correspondant au niveau majeur sont brisés, on peut affecter les nombres niveau mineur provisoires (NminP). Si une classe B dépend

d'une classe A, alors la classe B se verra affecter le nombre *niveau mineur* de la classe A augmenté de 1. A ce stade du processus d'intégration, les valeurs attribuées aux classes sont provisoires. Elles sont calculées d'abord au sein du niveau majeur sans tenir compte des dépendances inter niveaux. Ensuite, à partir des dits liens inter niveaux, nous affecterons un nombre niveau mineur définitif. En faisant référence à la figure 36, notre diagramme pour les nombre niveau mineurs (NminP) nous donnera à la figure 37 :

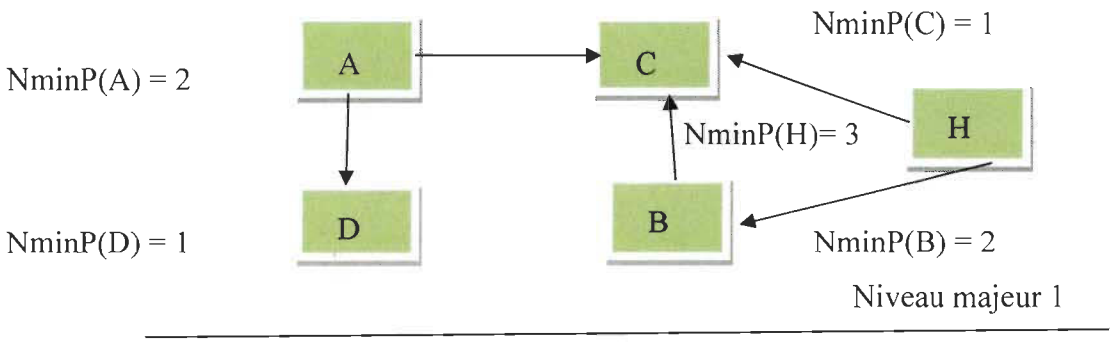


Figure 37 Obtention des nombres NminP pour le niveau majeur1

Nous remarquons que les classes A, B, C, D et H sont de *niveau mineur provisoire* (P) d'où

Pa = 2, Pb = 2, Pc = 1, Pd = 1 et Ph = 2

Pour le niveau 2, il n'y a pas de cycle à briser, une seule classe (classe G) s'y trouve, le nombre (NminP(g) = 2). Nous pouvons alors procéder à l'attribution des nombres niveau définitifs sous l'acronyme (NminD). Nous aurons alors :

Da = 2, Db = 2, Dc = 1, Dd = 1 et Dh = 3, Dg= 3

Ordonnancement des classes

Ainsi, le processus d'ordonnancement consiste à déterminer l'ordre d'intégration des classes. Il est basé sur les couples (*niveau majeur*, *niveau mineur*) déterminés précédemment. Les classes dont le niveau mineur est plus faible sont intégrées avant celles dont le *niveau mineur* est élevé.

CLASSE	Nombre niveau majeur	Nombre niveau mineur
A	1	2
B	1	2
C	1	1
D	1	1
H	1	3
G	2	3

TABLEAU 1 nombre niveau majeur et mineur des différentes classes

L'intégration des classes pour notre modèle sera :

D sera intégrée en premier

C sera intégrée en utilisant le bouchon A

A sera intégrée en utilisant la classe C et la classe D

B sera intégrée en utilisant la classe C

H sera intégrée en utilisant les classes C et B

G sera intégrée en utilisant les classes D et B

Intégration des aspects

Une fois l'ordre d'intégration des classes établi, on procède à l'intégration des classes avec l'intégration des aspects relatifs à chacune d'elles. D'une manière générale,

il s'agit, pour chacune des classes obtenues selon l'algorithme B3, de greffer les aspects pouvant être reliés à chacune d'elles. Plusieurs cas sont présents dans ce contexte : un seul aspect est relié à une classe donnée (A1), plusieurs aspects sont reliés à la classe (A1 et A11) et un aspect est lié à plusieurs classes (A2).

Selon la stratégie que nous préconisons, l'intégration des aspects doit se faire de façon incrémentale en fonction des critères relatifs à la complexité, au couplage et au type de dépendance. Ainsi, en se référant à la figure 33, l'aspect A2 est lié aux classes A, C et H. L'aspect A1 est lié à la classe A et l'aspect A11 est lié à l'aspect A1.

Selon l'ordonnancement obtenu précédemment, on commence par le test de la classe D. On procède par la suite à l'intégration de la classe C. Il se trouve que celle-ci est reliée à l'aspect A2 qui est relié à d'autres classes. On identifie le type de dépendance entre la classe C et A2, il s'agit du type As. Dans ce cas, on procède à l'intégration de la classe C avec l'aspect et le bouchon de test A. Si les autres dépendances de l'aspect avec le reste des classes sont de types It ou I, on définira des classes vides pour représenter celles-ci ; c'est les cas de la classe H. Le test de l'aspect A2 sera complet lorsque l'ensemble des classes auxquelles est associé l'aspect auront été intégrées.

On continue le processus de test en procédant à l'intégration de la classe A. Il se trouve que celle-ci est liée à deux aspects (A1 et A2). Selon nos critères (dépendances et complexité), on commence par l'intégration de l'aspect A1. Etant donné que l'aspect A1 est relié à un autre aspect A11, et sachant qu'un aspect ne peut être exécuté seul, on va intégrer les deux aspects A1et A11 à la classe A et procéder au test de celle-ci en utilisant la classe C et D.

Par la suite, on considère l'aspect A2. On identifie le type de dépendance. Il s'agit d'une dépendance de type C. On pourra donc intégrer l'aspect à la classe A et procéder au test de la classe en utilisant C et D, et on définira des classes vides pour

représenter les classes auxquelles l'aspect est lié; cas de la classe H.

On continue le processus d'intégration, en procédant à l'intégration de la classe B en utilisant la classe C. La classe B n'est reliée à aucun aspect.

Jusqu'à présent les classes qui ont été intégrées sont les classes D, C, A, B.

La prochaine classe à intégrer correspond à H. Elle est reliée à un seul aspect A2 (aspect relié à plusieurs classes). La dépendance est de type I (l'aspect utilise deux classes) qui relie les deux classes A et H. La classe A à été déjà testée, cela veut dire que l'aspect a été partiellement testé par rapport à ce type de relation. On peut donc procéder à l'intégration de l'aspect A2 à la classe H en utilisant C et B. Suite à cette intégration, le test (si on peut parler de test d'aspect) de l'aspect A2 est complet.

Enfin, on procède à l'intégration de la classe G en utilisant les classes D et B. La classe G n'est reliée à aucun aspect.

Le principal avantage que procure la stratégie que nous préconisons, est qu'elle n'augmente pas le nombre de bouchons de test (spécifique ou réel); le nombre de bouchons de test identifié lors de la première étape (avant intégration des aspects) reste le même.

Par ailleurs, étant donné que les aspects A1 et A2 se lient à une même classe A, on doit vérifier s'ils utilisent les mêmes point de jointure. Si c'est le cas, on doit procéder à différentes combinaison de ces deux aspects outre les critères de complexité, de couplage, et de dépendance qu'il faut considérer lors de leur intégration à la classe en question.

CHAPITRE 7

CONCLUSION ET PERSPECTIVES FUTURES

Nous avons présenté au début de notre document plusieurs stratégies d'intégration, selon différents auteurs, pour les systèmes orientés objet. Il faut noter, peu importe la stratégie, que le but visé lors de l'intégration des classes est de minimiser l'effort de test en réduisant en particulier le nombre de bouchons de test. Une des stratégies proposées dans ce contexte est la stratégie B3 développée par Badri et al. [Bad 05]. Il s'agit d'une stratégie basée sur un modèle qui dérive du diagramme de classes de conception et des diagrammes de collaboration entre classes UML. Cette stratégie offre de bons résultats, comparée à certaines stratégies existantes dans le domaine orienté objet, en termes de bouchons de test.

La stratégie que nous proposons, dans ce mémoire, pour supporter les tests d'intégration dans les systèmes orientés aspect correspond à une extension de la stratégie B3. La stratégie proposée est également basée sur les modèles UML (diagramme de classes, diagramme de collaboration entre classes). Elle tient compte aussi de la complexité des aspects, du couplage et des différents types de dépendances pouvant exister entre les classes et les aspects, établies à partir des constructions syntaxiques et sémantiques du langage AspectJ.

La stratégie décrite, dans ce mémoire, permet de maintenir les avantages offerts par l'algorithme B3 quant à la réduction du nombre de bouchons de test lors de l'intégration. En effet, la démarche adoptée utilise l'ordre d'intégration obtenu selon la

stratégie B3 dans un premier temps, puis considère chacune des classes selon l'ordonnancement obtenu pour lui intégrer les aspects qui lui sont reliés, s'il en existe. Nous avons procédé ainsi, car ce sont les aspects qui se greffent (se servent en quelque sorte) aux classes et non l'inverse. Ils ne peuvent, en fait, exister seuls sans les classes. Leurs liens sont unidirectionnels.

L'algorithme établit supporte une démarche d'intégration incrémentale. Une fois l'ordre d'intégration des classes déterminé, l'ordonnancement de l'intégration des aspects se fait en tenant compte de leurs dépendances, de leur complexité et de leur couplage.

Nous avons illustré et montré la faisabilité de notre stratégie à travers une étude de cas. Notre objectif (travaux futurs) étant de développer un environnement permettant de la supporter. Ceci pourra se faire relativement facilement, car plusieurs éléments de cet environnement existent déjà. Ils ont été développés dans le cadre de travaux connexes à ce projet. En particulier, nous disposons d'un outil supportant une stratégie de tests unitaires qui se concentre sur une classe à laquelle s'intègrent un ou plusieurs aspects (point de vue unitaire). Il s'agit, dans le futur, d'intégrer ces différents outils (développement d'un Framework global), pour pouvoir supporter notre approche et l'expérimenter à travers plusieurs études de cas réels.

BIBLIOGRAPHIE

[Aos 05] Aspect-oriented Software Development Web Site (AOSD). http:.aosd.net/

[Ale 04] Roger T. Alexander, James M. Bieman, Anneliese A. Andrews (2004), Towards the Systematic Testing of Aspect-Oriented Programs, Department of Computer Science, Colorado State University, Fort Collins, USA Technical report CS-4-105.

[Ajp 02] T AspectJ, The Aspect[tm] Programming Guide 2002.

[Ajw 05] AspectJ Web Site, http://eclipse.org/Aspectj/

[Anb 06] Prasanth Anbalagan, Tao Xie (2006) : Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs. Second Workshop on mutation Analysis 2006. Raleigh, NC, USA.

[Bad 04] L, Badri, M, Badri & V.S. Blé, Object-Oriented Testing: A method Level Approach, Proceeding of the 8th LASTED International Conference on Software Engineering and Applications, Cambridge, USA, nov 2004.

[Bad 05] L.Badri, M.Badri et V.S.Blé, A method level based approach for OO integration testing : An experimental study, Proceedings of the sixth ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing(SNPD2005), IEEE computer society press, Maryland, USA, may 2005, pp.102-109.

[Bal 04] D. Balzarotti and M. Monga, Using slicing to analyse aspect-oriented composition, 2004.

[Bal 01] J. Baltus, La programmation Orientée Aspect et AspectJ : Présentation et Application dans un système distribué, Mini-Workshop : Systèmes coopératifs. Matière approfondie, Institut d'informatique, Namur 2001.

[Bal 02] Baltus J., Institut d'informatique des FUNDP, La programmation orientée aspect et aspectJ : présentation et application dans un système distribué, http://www.info.fundp.ac.be/~ven/CISma/FILES/2002-baltus.pdf

[Bar 06] Bartsch , M and Harrison R., A Coupling Framework for AspectJ, Extended abstract, Proceeding of EASE 2006, Keele, UK. http://www.personal.rdg.ac.uk/~sir04mb2/Publication/bartsch-harrison-couplingFramework.pdf

[Bin 94] Robert V. Binder, Design for testability in object-oriented systems, Communication of the ACM, Vol37, Sept 1994, pp.87-100.

[Blé 05] Blé Stéphane Velou, Tests d'intégration des classes dans les systèmes orientés objet , Mémoire de maîtrise (2529), Université du Québec à Trois-Rivières, 2005.

[Bou 07] Bourque-Fortin, Maxime Génération et exécution de tests unitaires pour Les programmes orientéd-aspect , Mémoire de maîtrise, Université du Québec à Trois-Rivières, juillet 2007.

[Bri 99] Briand, L. C., Daly, J.W. and Wüst, J.K, (1999) A Unified Framework for Coupling Measurement in Object-Oriented Systems . IEEE Transactions on Software Engineering, 25(1), 91-120.

[Bri 02] L.C. Briand, J. Feng and Y Labiche, Using Genetic algorithms and Coupling Measures to Devise Optimal Integration Test Order, Proc Of the 14th ACM International Conference on Software Engineering and Knowledge Engineering, Itchier(Italy), July 2002, pp. 43-50.

[Bri 03] L.C. Briand, Y. Labiche and Y. Wang, An investigation of graph-based class integration test order strategies, IEEE Transactions on software Engineering, 29(7):594-607.

[Cec 04] Cecato, M and Tonella P., Measuring the effects of software Aspetisation, CD-Rom Proceedings of the first workshop of aspect Reverse Enginerring (WARE 2004), Nov 2004, Delft, The Nederlands.

[Cec 05] M.Cecato, P. Tonella, and F. Ricca. Is AOP code easier or harder than OOP code ? , In First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005), Chicago, Illinois, USA, march 2005.

[Han 03] Hanneman J., Chitchyan R. & Rashid A., Analysis of aspect-oriented software, AAOS 2003, Darmstadt, Allemagne.

[Har 92] Mary Jean Harrold, John D. McGregor and Kevin Fitzpatrick, Incremental Testing for Object-Oriented Class Structures, 14th international Conference on software Engineering, IEEE Computer Society, CA, May 1992, pp68-80.

[Hav 07] W. Havinga, I. Nagy, L. Bergmans and M. Aksif, A graph-based approach to modeling and detecting composition conflics related to introductions, In AOSD'07: Proceedings of the 6th international conference on aspect-oriented software developpement, pages 85-95, New-York, EY, USA, 2007ACM Press.

[Klm 97] Kickzales G., et al., Aspect-Oriented in proceeding of the European conference on object-oriented programming (ECOOP), Finland, 1997.

[Lad 02] Laddad R. JavaWorld,,I want my AOP! , Janvier 2002, [en ligne] http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html

[Lad 03] Laddad, Ramnivas," *AspectJ in action* ", Manning, Greenwitch,CT, 2003, 481 p.

[Lag 04] B.Lagaisse, W. Joosen and B. De Win. Managing semantic interference with aspect-integration contracts, In software Engineering Properties of Language and aspect technologies, 2004.

[Mah 04] Mark Mahoney, Atef Bader, Tzilla Elrad and Omar Aldawud: Using Aspects to abstract and modularize Statecharts, In the 5th Aspect-Oriented Modeling Workshop In Conjunction with UML, 2004.

[Mas 06] Philippe Massicotte, Test Orienté Aspect, Mémoire de maîtrise en mathématiques et informatique appliquées(2587), UQTR, 2006.

[Mce 05] N. McEachen and Alexander, Distributing classes with Woven concerns. An exploration of potential scenarios. Proceedings of the 4th international

Conference on Aspect-oriented software developpement, pp 192-200, Chicago, Illinois, USA, March 14-18, 2005.

[Mcg 94] John D. McGregor and Tin Korson,Integrating Object-Oriented Testing and Developpement Processes, Communication of the ACM, Vol 37 (9), Sept 1994, pp.59-77.

[Mil 02] A. Milanova, A. ROUNTEV and Barbara G. Ryder, "Precise Constructing Object Relation Diagrams", International Conference on Software Maintenance, ICSM'02, October 2002.

[Mor 04] M. Mortensen and Alexander, Adequate testing of aspect-oriented Programs, Technical report cs 04-110, Colorado state University, Fort Collins, Colorado,USA, December 2004.

[Omg 00] Object Management Group (OMG), Unified Modeling Language Specification, Version 1.3, Mar 2000.

[Reg 07] Reginaldo Ré, Otâvio Augusto Lazzarini, Paulo Cesar Masiero, Minimising Stub Creation During Integration Test of Aspect Oriented Programs, Workshop WTAOP '07, March, Vancouver, Canada , 2007.

[Sou 01] Neelan Soundarajan and Benjamin Tyler, Specification-Based Incremental Testing of object-oriented Systems , IEEE, 2001.

[Tai 99] K.C. Tai and F.J. Daniels, Interclass Test order for Object-Oriented Software , Journal of Object-Oriented Programming, vol 12(4), 1999.

[Tar 72] R. Tarjan,Depth-first search and linear graph algorithms, SIAM J. Comput., Vol.1, n2, June1972, 146-160. ISSN 1064-8275.

[Tes 04] Tessier,F. Badri,M. Badri, L. A model-based detection of conflicts between crosscutting concerns : Towards a formal approach , In: International workshop on Aspect-Oriented Software Developpement, 2004.

[Tra 00] Y. Le Traon, J.M. Jézéquel and P.Morel, Efficient Object-Oriented Integration And Regression Testing , IEEE Transactions on Reliability, vol. 49(1), March 2000, pp.12-25.

[Tri 02] Triskell, le Han, *Test et modèle UML :* Stratégie, plan et synthèse de Test , Phd thesis, Ecole doctoraleMatisse, Université de Rennes1, 2002.

[Vid 01] Cristina Videira Lopes and Trung Chi Ngo. UnitTesting Aspectual Behavior.

[Vie 05] Nicolas Viel, Présentation de l'AOP, exemple d'utilisation en C#, http://www.labo-dotnet.com/Articles/Divers/Pr%C3%A9sentation%20de%20l%20AOP,%20exemple%20d%20utilisation%20en%20Csharp/1dot%20Pr%C3%A9sentation%20de%20l%20AOP/1/1532.aspx, 28 juin 2005.

[Wal 99] Walker R., Baniassad E and Murphy G., An initial assessement of Aspect-Oriented Programming, in Proceedings of the 21th

International Conference on Software Engineering, Los Angeles,CA, 1999.

[Xat 03] Xerox AspectJ Team Eclipse.org, *Aspect programming guide*, 2003.

[Xu 01] Dianxiang Xu, Weifeng Xu, Kendall Nygard (2005). A State-Based Approach to testing Aspect-Oriented Programs. In Proc of the 17th International of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), July 14-16, Taiwan.

[Xu 02] Weifeng Xu, Dianxiang Xu, Vivek Goel and Ken Nygard. Aspect Flow Graph For Testing Aspect-Oriented Programs. Proceedings of the 8th IASTED International Conference on Software Engineering and Applications. Oranjestad, Aruba (Caribbean), August 29-31, 2005.

[Xu 05] Dianxiang Xu, Weifeng Xu. Vivek Goel and Ken Nygard: Aspect Flow Graph For Testing Aspect-Oriented Programs, Proceeding of the 8th IASTED International Conference on software Engineering and Applications, Oranjestad, Aruba (Caribbean), august 29-31, 2005.

[Xu 06a] Guoqing Xu,*A* Regression Tests Selection Technique for Aspect-Oriented Programs , In Workshop on Testing Aspect-Oriented Programs, pages 15-20, 2006.

[Xu 06b] Weifeng Xu and Dianxiang Xu: State-Based Testing of Integration Aspects , In Proceeding of the second workshop on Testing Aspect-Oriented Programs (WTAOP 06), Juillet 2006, Maine, USA.

[Zha 01] Jianjun Zhao (2003). Data-flow-based unit testing of aspect-oriented programs. In Proc. 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003), pp.188-197. Dallas, Texas, USA

[Zha 02] Jianjun Zhao (2002). Tool Support for Unit Testing of Aspect oriented Software OOPSLA2002 Workshop on tools for Aspect oriented Soft-Ware Developpement, Seatle, WA, USA.

[Zha 03] Zhao, J., Data-flow based unit testing of aspect-oriented programs, Proceding of the 27th annual conference software and application, 2003.

[Zha 04] J. Zhao, Measuring Coupling in Aspect-Oriented System, 2004, http://cse.sjtu.edu.cn/~zhao/pub/pdf/metrics04lbp.pdf#search=%22measuring%20coupling%20in%20aspect-oriented%20systems%22.

[Zho 04] Y. Zhou,D. Richardson, and H. Ziv, Towards a practical Approach to test aspect_oriented software, In Proc. 2004 Workshop on testing Component-based Systems (TECOS 2004), Net.ObjectiveDays,September 2004.