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
MAXIME BOURQUE-FORTIN

GÉNÉRATION ET EXÉCUTION DE SÉQUENCES DE TESTS UNITAIRES POUR LES PROGRAMMES ORIENTÉS ASPECT

JUILLET 2007

GÉNÉRATION ET EXÉCUTION DE SÉQUENCES DE TESTS UNITARES POUR LES PROGRAMMES ORIENTÉS ASPECT

Maxime Bourque-Fortin

SOMMAIRE

La programmation orientée aspect est un paradigme émergeant du génie logiciel. Elle permet essentiellement de mieux représenter les préoccupations transverses dans un programme. Cependant, malgré les nombreux avantages que semble offrir le paradigme aspect, il ne reste pas moins qu'il n'est pas encore mature. Il introduit de nouvelles dimensions en termes de contrôle et de complexité. Les aspects ont, en fait, une grande latitude pour interagir avec les classes de base d'un système. Ces interactions constituent une nouvelle source d'erreurs dans les programmes. Les techniques actuelles de test orienté objet ne sont pas adaptées pour le test des programmes orientés aspect. De nouvelles techniques de test doivent donc être développées pour les programmes orientés aspect. Les aspects, de part leur nature, ont la capacité de s'exécuter en s'insérant dans le flôt de contrôle du programme, selon les contextes spécifiés et sans qu'aucune référence n'y soit faite dans le code des classes de base. Nous proposons, dans ce travail, une nouvelle technique de test unitaire orienté aspect basée sur le comportement dynamique. Nous focalisons sur l'intégration d'un ou plusieurs aspects à une classe. Notre objectif est de vérifier que cette intégration se fait correctement, sans altérer le comportement original de la classe de base. Nous introduisons plusieurs critères de test afin de déterminer ce que les tests doivent couvrir. Nous avons, par ailleurs, développé un outil de génération et d'exécution de tests supportant l'approche proposée et intégrant les critères de test développés.

GENERATING AND EXECUTING UNIT TESTING SEQUENCES FOR ASPECT-ORIENTED PROGRAMS

Maxime Bourque-Fortin

ABSTRACT

Aspect-Oriented Programming is a new Software Engineering Paradigm. It improves essentially the representation of crosscutting concerns in a program. However, even after considering the numerous advantages that the Aspect paradigm seems to offer, it is still not mature yet. It introduces, in fact, new dimensions in terms of control and complexity. Moreover, aspects have great latitude to interact with the basic classes of a system. Those interactions constitute a new source for faults in programs. Existing object-oriented testing techniques are not adequate for testing aspect-oriented programs. New testing techniques must then be developed for aspect-oriented programs. We propose, in this work, a new technique for aspect-oriented unit testing based on dynamic behavior. We focus on the integration of one or more aspects to a class. Our approach is based on UML Statechart Diagrams. The primary objective is to verify that this integration is done correctly, without modifying the original behavior of the basic class. We introduce several testing criteria in order to determine what the tests must cover. Moreover, we developed a tool supporting the proposed approach for the generation and the execution of tests for the aspect-oriented paradigm integrating the developed testing criteria.

REMERCIMENTS

Sans la contribution de plusieurs personnes, ce projet n'aurait pas été possible. Je tiens à les remercier pour leur soutien et leur aide.

Premièrement, j'aimerais particulièrement remercier mes co-directeurs Mourad Badri et Linda Badri, professeurs au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières, qui ont rendu possible, grâce à leur expertise et leur soutien, la réalisation de ce travail.

Je voudrais aussi remercier mes parent Louise et Gaston ainsi que ma copine Geneviève pour leur soutien tout au long de ma maîtrise. Sans vos encouragements, rien de ceci n'aurait été possible.

J'aimerais aussi remercier tous mes collègues du laboratoire de génie logiciel ainsi que les membres du département de mathématiques et informatique pour leurs conseils et leurs suggestions.

TABLE DES MATIÈRES

Page

LISTE DES FIGURES

Page

LISTE DES TABLEAUX

Page

INTRODUCTION

La programmation orientée aspect est un nouveau paradigme du génie logiciel. Elle suscite, de plus en plus, l'intérêt des chercheurs. Plusieurs langages orientés aspect émergent actuellement. AspectJ, le plus connu de ces langages, est une extension orientée aspect du langage Java. Les langages de programmation orientée objet éprouvent, en effet, des limites sérieuses quant à la représentation des préoccupations transverses dans un programme. Le code correspondant à ces préoccupations est souvent dispersé et dupliqué dans plusieurs classes de base. Cette dispersion rend le code des programmes difficile à comprendre, à tester, à maintenir et à réutiliser. Le paradigme aspect permet, en fait, de regrouper le code de ces préoccupations dans des unités modulaires appelées aspects. Ceci se traduit par une meilleure modularité au niveau du code avec tous les avantages qui en découlent. Le paradigme aspect introduit de nouvelles abstractions permettant de mieux représenter les préoccupations transverses. Les aspects permettent de créer des modules cohésifs implémentant des préoccupations spécifiques qui, autrement, auraient été distribuées (dispersées) à travers plusieurs abstractions de base. Cette nouvelle façon de concevoir les programmes procure plusieurs avantages parmi lesquels nous pouvons citer une meilleure modularité et cohésion des composants, une séparation des préoccupations plus claire et une meilleure localisation des modifications. De cette façon, les abstractions primaires sont construites de façon plus cohésive puisque leur implémentation se concentre sur leurs responsabilités premières. Le code initialement éparpillé dans les classes d'un programme orienté objet a sa propre structure grâce aux aspects.

Cependant, malgré les nombreux avantages que semble offrir la technologie aspect, il ne reste pas moins qu'elle est loin d'être mature. Les aspects introduisent de nouvelles dimensions en termes de contrôle et de complexité. Ils ont, en fait, une grande latitude d'interaction avec les classes de base d'un système. Ces interactions constituent une nouvelle source d'erreurs dans les programmes tels que mentionnés par plusieurs

auteurs. Les techniques actuelles de test orienté objet ne sont pas adaptées pour le test des programmes orientés aspect. Elles ne couvrent pas les nouvelles dimensions introduites par les aspects, en particulier, le comportement des aspects et leurs interactions avec le reste des modules du programme. Les aspects, de par leur nature, ont la capacité de s'exécuter en s'insérant dans le flot de contrôle du programme selon les contextes spécifiés, sans qu'aucune référence soit faite (explicitement) dans le code des classes de base. Cette exécution se fait à leur insu. Par ailleurs, un aspect seul n'a pas d'identité propre. Il ne peut être testé isolément. Son exécution est toujours tributaire du contexte auquel il s'intègre. La relation entre un aspect et les classes auxquelles il se greffe est fondamentalement différente de celle qui existe entre les classes dans un système orienté objet. Ceci indique clairement qu'il faut définir de nouvelles techniques de test appropriées à cette nouvelle façon de construire les programmes. Ces techniques doivent, donc, tenir compte des nouvelles dimensions en termes de contrôle introduit par les aspects et des nouvelles sources d'erreurs qu'elles engendrent. C'est dans ce contexte et dans un objectif d'assurance-qualité que nous avons décidé de nous intéresser à ces problèmes.

CHAPITRE 1

ÉTAT DE L'ART

Depuis le début des années 2000, de nombreux travaux portant sur les « aspects » ont été publiés, mais très peu (relativement) se sont penchés sur le test des systèmes orientés aspect. Après des recherches sur internet ainsi que les librairies de l'ACM et de l'IEEE, on s'aperçoit que très peu de textes portent sur le test dans le paradigme aspect et que la majorité s'oriente plutôt vers l'utilisation de la technologie aspect à des fins de test [Alexander01] [Xu01]. Malgré cela, plusieurs auteurs ont vu les nouveaux défis qu'apportait ce paradigme en plus de ses avantages. Alexander, Bieman et Andrews présentent dans l'article [Alexander01] les caractéristiques propres à la technologie aspect qui font que ce nouveau paradigme pose quelques problèmes au niveau du test. Par la suite, les auteurs nous proposent un modèle de fautes (fault model) pour le paradigme aspect en commençant par désigner quatre sources d'erreurs dans les systèmes orientés aspect :

- Les erreurs produites par le code objet classique.
- Les erreurs produites par le code interne des aspects.
- Les erreurs produites par l'introduction du code aspect dans le code objet.
- Les erreurs produites lors de l'introduction du code aspect de plusieurs aspects dans une même portion de code objet.

L'article nous présente ensuite six types d'erreurs issus de ces quatre sources qui forment une première tentative de modèle de fautes pour le paradigme aspect. Après avoir présenté leur modèle, les auteurs nous proposent une série de critères de test pour couvrir le modèle de fautes qu'ils ont présenté :

- Test de la force des pointcuts
- Test de l'ordonnancement des aspects
- Retester l'ensemble des postconditions des méthodes étendues par un ou des aspects
- Retester l'ensemble des invariants des états pour chaque méthode touchée
- Erreurs reliées au flot de contrôle (intégration)
- Test des erreurs dans les dépendances de contrôle.

Cet article est l'un des premiers qui ont tenté d'établir préalablement des critères de test pour déterminer ce qui doit être testé et en quelles circonstances selon les spécificités du paradigme aspect avant de développer et proposer une méthode de test.

Les articles [Zhao01], [Zhao02] et [Zhao03] nous présentent une méthode de tests unitaires pour les classes et aspects d'un système orienté aspect. La méthode est basée sur les tests par les diagrammes de flot de contrôle. La méthode de test que présentent les articles se trouve à trois niveaux : intramodule (le test des méthodes, advices, introductions, etc.), intermodules (le test des interactions entre les différents modules) et intraclasse/intraaspect (le test d'une classe ou d'un aspect dans son ensemble). Cette méthode se veut ainsi tester unitairement les aspects au même titre que les classes. Cependant, il faut être conscient que les aspects ne possèdent pas d'interface publique au même titre que les classes, c'est à travers l'exécution de leurs advices lors des cassures du flot de contrôle dans le code objet qu'ils s'exécutent. Ces cassures se font à différents endroits et à différents moments de l'exécution du programme, donc dans différents contextes. Les pointcuts, qui définissent les endroits où intervenir dans le flot de contrôle, capturent une partie du contexte pour la passer à travers les paramètres de l'advice qui est exécuté. Mais, le résultat de l'exécution de cet advice peut être extérieur à l'aspect ou à la tranche du contexte qui lui est passé. Donc, le bon fonctionnement

d'un module, tel qu'un advice, peut très bien être inconnu si on reste dans les limites du module ou de la tranche de contexte qu'il connaît. Dans certains cas, on ne peut tester un module seul, son bon fonctionnement sera toujours tributaire du contexte dans lequel il s'exécute, et sans la connaissance de ce contexte particulier il sera impossible de savoir s'il y a des erreurs. Par ailleurs, et même s'il fonctionne correctement pour ce contexte particulier rien ne garantit qu'il fonctionnera pour tous les contextes. L'article [Zhao02] explique que la méthode consiste à générer des séquences de test aléatoires pour couvrir l'ensemble des chemins et cela pour les aspects autant que pour les classes. 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 [Alexander02] l'a souligné, les aspects n'ont pas d'existence propre. C'est donc à travers leur influence dans les différents contextes qu'ils prennent une forme tangible. Si on se rapporte aux quatre sources d'erreurs des systèmes orientés aspect présentées dans l'article [Alexander01], on voit qu'ils ont identifié l'introduction du code aspect dans le code objet, donc dans un contexte, comme l'une des sources d'erreurs. Donc, en évacuant du test de l'aspect le contexte dans lequel il sera exécuté, il échappe un des types d'erreurs propres au paradigme aspect. De plus, l'article [Alexander02] nous rappelle qu'une des caractéristiques fondamentales des aspects est qu'ils sont fortement liés aux contextes dans lesquels ils évoluent. Il s'agit justement d'une des difficultés qui se dressent devant le développement d'une méthode de test aspect et donc que l'on ne peut ignorer. L'article [Zhao04] présente une technique de sélection de tests pour produire des tests de régression dans un système orienté aspect basé sur le flot de contrôle dans les systèmes aspect.

L'article [Xu01], quant à lui, nous propose une technique basée sur les diagrammes d'états. Il s'agit de l'extension de la méthode FREE (Flattened Regular Expression) pour le test dans le paradigme objet. L'article nous présente une adaptation de la

représentation graphique FREE pour tenir compte des aspects. Il s'agit de l'Aspectual State Model (ASM) qui permet de représenter dans un diagramme d'états les ajouts faits par les aspects. À partir de ces ASM, il propose une extension de la méthode FREE pour la génération d'un arbre qui servira à générer la suite des chemins à parcourir afin de couvrir l'ensemble des possibilités. Nous obtenons ainsi des séquences de test qui permettent de tester l'ensemble des possibilités de ce diagramme. En deuxième partie, l'article nous présente les diagrammes d'Aspect Flow Graph (AFG) qui permettent de représenter les flots de contrôle des transitions entre les différents états du ASM en tenant compte de l'influence des aspects. La technique présentée ici est très intéressante, mais se limite à tester des ensembles classe/aspects en un seul bloc et non de façon modulaire. Le diagramme représente le comportement de la classe et de tous les aspects qui la touchent comme une nouvelle entité. Si un nouvel aspect est ajouté ou modifié, il faut retester le tout car le ASM sera modifié et une nouvelle série de séquences de tests sera générée qui couvrira l'ensemble du nouveau ASM. Par exemple, voici un diagramme ASM extrait de l'article [Xu01] :

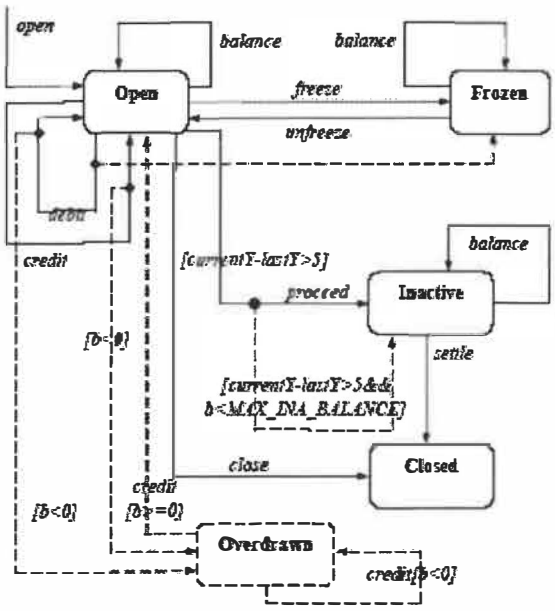

Figure 1 Extrait de l'article [Xu01]

Les lignes pleines représentent les éléments de la classe et ceux en pointiller les éléments correspondant aux aspects. Si, par exemple, nous ajoutons un nouvel aspect qui permet de faire le suivi de l'information envoyée aux clients lorsque leur compte est inactif, il y aura des éléments ajoutés autour de l'état Inactive, mais ces modifications ne toucheront pas le reste du diagramme. Dans un tel cas, la technique présentée génère un nouvel ensemble de séquences de test qui couvriront l'ensemble du diagramme et non uniquement ce qui est concerné par cette modification, car cette technique teste un diagramme comme étant une seule entité et non un regroupement. Cet article ne fait jamais référence aux introductions d'aucune sorte. Il ne semble prendre en compte que les mécanismes des pointcuts et des advices du paradigme aspect. Si ce dernier élément est une limite de la méthode, cela signifie que la méthode ne couvre pas l'ensemble des mécanismes des langages aspects courants et ne peut donc couvrir l'ensemble des sources d'erreurs du paradigme.

L'article [Xu02], quant à lui, nous présente une méthode de test unitaire basée sur le flot de contrôle. L'article nous présente une approche qui veut fusionner les diagrammes d'états des classes ainsi que les diagrammes d'état des aspects dans un même diagramme aspect nommé Scope State Model (ASSM) comme le montre la figure 2 :

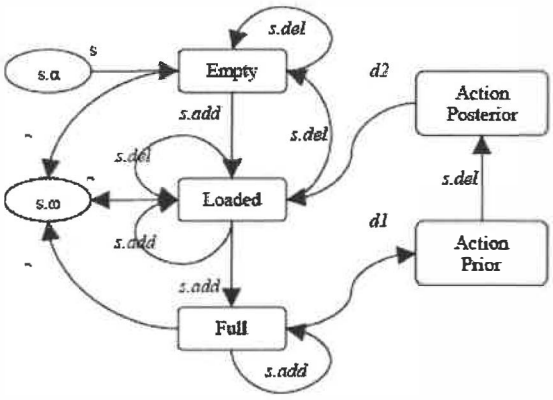

Figure 2 ASSM pariel extrait de [Xu02]

Ce diagramme représente un ASSM d'une classe pile. Il ne s'agit que d'un diagramme partiel, car seules les actions des advices before et after sur la seule méthode delete faisant passer la classe de l'état Full à Loaded. Ensuite, on y insère les diagrammes de flot de données des advices et méthodes pour obtenir un Aspect Flow Graph (AFG) :

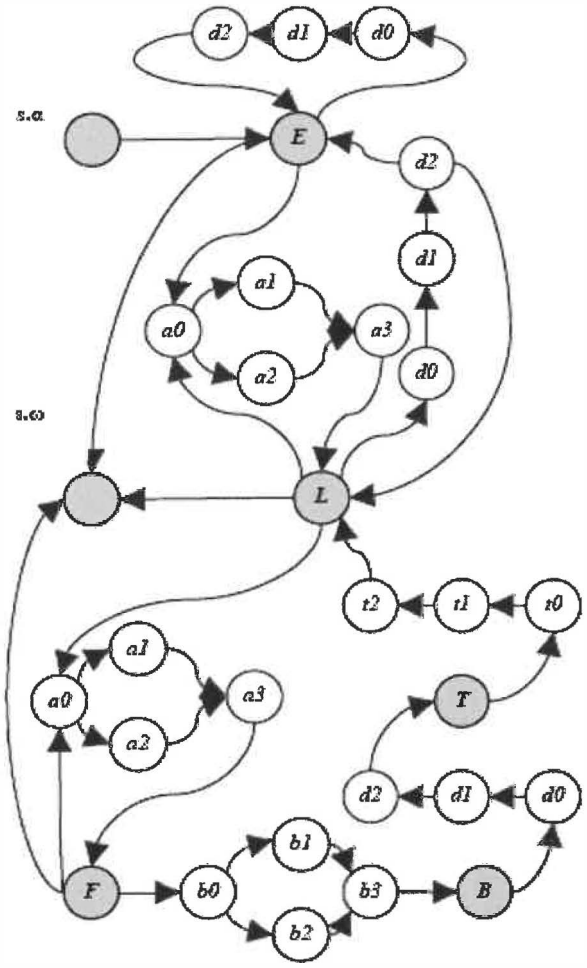

Figure 3 AFG partiel extrait de [Xu02]

Ce diagramme représente le même exemple. La technique consiste à remplacer les liens entre les différents états par la portion du diagramme de flot de contrôle représentant la méthode concernée. L'advice before est représenté par la lettre b, after par t et delete par d. Ainsi, on reconnaît la méthode delete Full->Loaded avec ses advices du ASSM. De ce dernier diagramme, il tire ses séquences de test après en avoir extrait un arbre. Les auteurs soulignent, lors de leur couverture de la littérature, que la majorité des auteurs qui se sont penchés sur la problématique ont peu porté leurs efforts sur l'identification de critères de test. Cependant, l'article [Xu02] ne définit pas lui non plus de critères de test autres que de simples vérifications booléennes pour savoir s'il existe un lien entre

telle classe et tel aspect. Il semble également que cette approche ne tient compte que des mécanismes d'advices before et after. Comme l'article [Xu01], dans ce cas la méthode ne peut couvrir l'ensemble des sources d'erreurs propres à ce paradigme qui justifient justement la création de nouvelles méthodes destinées à couvrir ces particularités.

Parmi ces différents articles, certains nous présentent 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 aspect. Dans ce dernier type d'article, nous comptons entre autres les articles [Zhao01], [Zhao02], [Zhao03], [Xu01] et [Xu02]. Comme nous l'avons présenté, la méthode de Zhao ne semble pas tenir compte des particularités des aspects et les teste comme des classes à peine particulières. Une telle méthode ne peut, selon nous, couvrir avec efficacité l'ensemble des erreurs d'un système orienté aspect. Les articles [Alexander01] et [Alexander02] vont dans le même sens si on regarde les caractéristiques des aspects et les différentes sources d'erreurs propres aux systèmes aspects qu'ils présentent. C'est pourquoi nous allons vers une méthode de test qui prend en compte ces caractéristiques. Nous avons opté pour une méthode de test qui s'oriente vers le test des aspects dans le contexte d'une intégration aux classes auxquelles ils sont reliés.

Une autre approche intéressante est proposée dans [Mahoney01]. Dans cet article, les auteurs présentent un framework de génération de code permettant la gestion des préoccupations transversales à l'aide des diagrammes d'états. Ce framework n'utilise pas un langage orienté aspect tel qu'AspectJ ou Hyper/J. Tout est réalisé dans le paradigme orienté objet. Ce qui est particulièrement intéressant dans le cas de cet article, c'est l'utilisation des diagrammes d'états pour la modélisation des préoccupations transversales à l'aide de différentes zones dans les diagrammes et les liaisons entre les transitions de chaque zone pour représenter le comportement des advices et pointcuts des aspects.

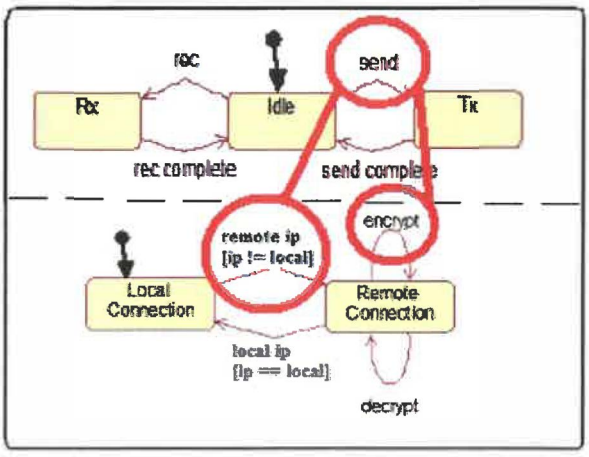

Figure 4 Diagramme d'états-transitions représentant une classe de communication et un aspect d'encryptions dans une zone orthogonale [Mahoney01]

Par exemple, Mahoney présente dans l'article un aspect qui permet l'encryption lors de communications extérieures qui est associé à une classe de communication. On peut voir l'association entre la transition « send » de la classe et les transitions « remote ip » et « encrypt » de l'aspect qui représentent, en fait, le comportement des pointcuts et advices de l'aspect associés à la méthode « send » de la classe. Ce travail permet de voir comment les diagrammes d'états ont la capacité de représenter le comportement des aspects et leur influence sur les classes.

Certains travaux portant sur le test unitaire dans le paradigme orienté aspect ont développé leur propre framework de système aspect, car les plus répandus comme AspectJ ne permettent pas des tests unitaires sur les aspects. L'article [Videira01] prend cette approche. Dans cet article, les auteurs expliquent que la structure même des langages tel qu'AspectJ ne peut supporter des tests unitaires sur des aspects totalement hors de tout contexte d'intégration à des classes. La principale cause étant le fait que dans ces systèmes les aspects ne peuvent pas être instanciés. Il ne s'agit pas d'élément que l'on peut créer. D'autres éléments sont aussi cités comme le manque d'ouverture de la réflexion des aspects, mais il s'agit plus de détails techniques que de problèmes

fondamentaux liés à la nature des aspects contrairement au premier. C'est pour cela qu'ils ont développé Jaml (Java Aspect Markup Language) où les aspects sont des entités instanciables. Le reste de l'article présente JamlUnit qui est leur système de test unitaire des aspects dans le système Jaml.

D'autres travaux portent sur d'autres aspects des tests unitaires dans un environnement orienté aspect. Par exemple, l'article [Anbalagan01] présente une technique de test de mutation de pointcut. La technique de test de mutation classique consiste à générer automatiquement des mutants des classes qui sont sous test (tests unitaires) et à vérifier si ces altérations (qui sont presque systématiquement des erreurs) seront détectées par les tests unitaires. Si les tests couvrent adéquatement les classes, alors ces mutations seront détectées, dans le cas contraire où ces mutations passeraient inaperçues cela indiquerait que les tests unitaires ne couvrent pas suffisamment les classes. L'idée de l'article [Anbalagan01] est de faire des mutations de pointcut afin d'éprouver les tests par rapport à la portée trop grande ou trop petite des pointcuts des aspects. L'article [Alexander01] avait cité ce point comme une des sources possibles d'erreurs dans les systèmes orientés aspect.

CHAPITRE 2

TECHNOLOGIE ASPECT

Le paradigme aspect reprend les éléments et principes du paradigme objet, mais en lui ajoutant de nouveaux éléments qui permettent de nouveaux comportements ainsi que de nouvelles façons de construire les systèmes. Le but premier du paradigme aspect est de regrouper, dans des entités modulaires, les concepts qui entrecoupent plusieurs tâches d'un système et qui sont habituellement dispersés dans plusieurs classes. Ces entités sont les aspects. Ainsi, dans un système aspect, les classes ainsi que les interfaces existent avec les mêmes mécanismes d'héritage et de polymorphisme. Un aspect peut contenir les mêmes éléments qu'une classe à l'exception des constructeurs, mais c'est dans les éléments qui leur sont propres que les aspects se démarquent des classes. Les aspects introduisent les concepts de points de jointure, points de coupure, d'advices et d'introductions. Ce sont les comportements de ces éléments qui sont responsables de la différence des aspects face aux classes.

2.1 Point de jointure

Un point de jointure (appelé par la suite jointpoint) est un endroit bien précis dans le code objet, ce peut être un appel de méthode ou encore l'accès à un attribut par exemple. Ces jointpoints sont définis dans les points de coupure (appelés par la suite pointcuts) qui sont eux-mêmes des éléments définis dans le corps des aspects.

2.2 Point de coupure

Ainsi, un pointcut définit un ou plusieurs jointpoints. Ces pointcuts servent à définir le champ d'action des advices qui leur sont associés. Les pointcuts sont composés d'une

série de spécifications qui définissent l'ensemble des jointpoints qu'ils contiennent. Nous avons, dans ce qui suit, un exemple de pointcut qui regroupe l'ensemble des appels des méthodes publiques de la classe Figure peu importe le type de retour ou les paramètres des méthodes :

```
pointcut FigurePublicCall : call(public * Figure.* (..));
```

2.3 Advices

Les advices, quant à eux, sont un type de méthode qui contient du code mais dont la méthode d'intervention diffère d'une méthode classique. Chaque advice est associé à un ou une composition de pointcuts qui définissent où et quand il interviendra dans le cours de l'exécution du système. Ainsi, les pointcuts définissent les endroits où une cassure du flot de contrôle sera faite pour passer le contrôle aux advices. Donc, les advices s'exécutent sans que le contrôle ne leur soit jamais passé « volontairement » mais en opérant des cassures dans le flot de contrôle à l'insu de celui qui le possède. Il existe trois types d'advice : before, after et around.

Les advices de type before permettent d'exécuter le code qu'ils contiennent juste avant l'arrivée du contrôle au point spécifié par le pointcut de l'advice. Après l'exécution de l'advice, le contrôle revient invariablement au point où il avait été détourné et reprend son cours normal. Nous avons ici un exemple d'advice de type before qui se déclenche avant chaque jointpoint du pointcut FigurePublicCall :

```
before( ) : FigurePublicCall( ) {
        System.out.println("Avant un appel d'une méthode publique de Figure.");
}
```

Les advices de type after sont très semblables à ceux de type before à l'exception que le code contenu dans l'advice s'exécute après que le contrôle soit passé à travers le segment de code spécifié par le pointcut de l'advice. Il existe quelques variantes de l'advice after qui n'existent pas pour ceux de type before, mais ce ne sont que des conditions d'exécution qui ne changent pas sa nature. Ces conditions sont de deux types. Le premier est conditionnel à la nature de l'élément qui est retourné par la méthode concernée. L'autre type est conditionnel à la levée d'un certain type d'exception lors de l'exécution du code pointé par le pointcut. Mais, dans son ensemble, il s'agit du même concept que l'advice de type before à l'exception de son moment d'exécution. Voici un exemple d'advice after qui se déclenche lorsqu'un jointpoint du pointcut FigurePublicCall retourne une exception :

```
after() throwing (Exception e): FigurePublicCall() {
        System.out.println("Exception levee : " + e);
}
```

Les advices de type around prennent le contrôle au même moment que ceux de type before, soit juste avant l'exécution du code pointé par le pointcut de l'advice. Cependant, ce type d'advice diffère des deux autres types d'advices par le fait que l'advice s'exécute à la place de la méthode appelée et que l'exécution de la méthode d'origine est déclenchée manuellement dans le code de l'advice à n'importe quel moment, ce qui n'inclut jamais. Alors que pour les types d'advice before et after elle était exécutée invariablement avant ou après l'exécution du code de l'advice. Il s'agit d'une caractéristique qui met les advices de type around en marge des deux autres types. L'utilité des advices around est qu'on peut les utiliser pour mettre dans un seul advice un travail qui demande l'exécution de code avant et après un pointcut plutôt que d'avoir deux advices, un before et un autre after, pour faire ce même travail qui est lié et ainsi amélioré la cohésion de l'advice. Nous avons dans ce qui suit un exemple d'un advice around qui se déclenche sur tous les appels des méthodes publiques de la classe Figure

qui retourne une valeur de type int et prend un seul paramètre de type int. Cet advice exécute la méthode d'origine avec le mot clé proceed mais en doublant la valeur du paramètre fourni lors de l'appel de la méthode d'origine :

```
int around (int i) : call( public int Figure.*(int)) && args(i) {
        return proceed(i*2);
}
```

2.4 Introductions

Si les pointcuts et advices se situent au niveau des modifications dynamiques du code objet, les introductions elles influencent le code objet de façon statique. Les introductions se divisent en trois types : les introductions d'éléments, de structures et de gestion. Les introductions d'éléments permettent d'ajouter à une ou plusieurs classes de nouveaux attributs ou méthodes de façon publique à tous ou plus habituellement interne à l'aspect. Ici, nous avons l'ajout d'un attribut à la classe figure de façon locale à l'aspect :

```
private int Figure.NouveauInt = 0;
```

Les introductions de structures quant à elles permettent d'ajouter à une ou plusieurs classes soit une nouvelle classe mère ou de nouvelles interfaces à implémenter. Ces introductions de structures sont toujours publiques, c'est-à-dire que ces nouveaux liens seront visibles de tous. Dans les cas où il est nécessaire d'étendre les classes ainsi modifiées pour respecter les nouvelles exigences qu'imposent ces introductions, l'aspect devra également introduire les éléments nécessaires de façon publique. Nous avons ici un exemple d'une introduction de structure qui ajoute une interface à implémenter à la classe Figure :

```
declare parents: Figure implements Drawable;
```

Les introductions de gestion permettent de gérer certains éléments du système. Cette gestion prend plusieurs formes. Elle peut se faire en levant des erreurs ou avertissements de façon statique lorsque certaines conditions sont remplies, en permettant de contourner et gérer manuellement les erreurs statiques du langage objet ou en définissant des priorités entre les différents aspects dans un système. Voici un exemple où un avertissement est levé lorsqu'un jointpoint du pointcut PublicFigureCall est atteignable :

declare warning: PublicFigureCall : "Le pointcut PublicFigureCall est atteignable";

2.5 Différences entre la programmation objet et aspect

Nous avons vu les différents mécanismes, autant dynamiques que statiques, qui sont les caractéristiques des aspects et qui leurs permettent d'agir dans les systèmes orientés aspect. Il est à noter que certains de ces mécanismes, comme les advices around ou encore les introductions de structures, peuvent avoir des effets très importants sur les classes et que ces effets peuvent très bien être contraires à la philosophie d'une bonne modélisation de système et qu'il faut donc être prudent avec ces mécanismes d'un paradigme pas encore mûr.

Mais là où est la véritable particularité des aspects, c'est au niveau du mode d'intervention dans le déroulement du système que leur permettent ces mécanismes. Alors qu'une classe ne peut être exécutée que lorsqu'on lui fait référence de façon explicite dans le déroulement du programme, un aspect lui interviendra par lui-même lorsqu'il le devra pour assurer le respect de ses spécifications. Donc, les aspects produisent des cassures, selon leur définition, dans le flot de contrôle du programme pour intervenir lorsqu'ils le doivent plutôt que lorsqu'une autre entité leur passe le contrôle. Ainsi, nous avons du code objet qui est complet en lui même pour remplir son rôle mais qui est étendu à son insu par les aspects. Nous pouvons voir que les aspects

apportent une nouvelle dimension au niveau du flot de contrôle dû à leur mode d'intervention. Cette nouvelle dimension est surtout soulignée par l'insertion de code aspect au cours de l'exécution du code objet sans que ce dernier n'ait été conçu en conséquence pas plus que le code aspect n'ait été conçu pour ce contexte particulier. Nous mélangeons ainsi deux codes qui ont chacun été conçus pour répondre à leurs préoccupations propres et non pour se compléter et remplir une même préoccupation. Il y a quatre caractéristiques des aspects qui sont présentées dans l'article [Alexander01] et qui sont responsables de ces différences avec les classes :

- Les aspects n'ont pas d'identité ou d'existence propre
- L'implémentation des aspects peut être hautement couplée au contexte auquel ils sont noués [woven]
- Les dépendances des données et du contrôle ne sont pas visibles directement du code source des aspects ou des classes
- Les effets secondaires produits par un mauvais ordre d'exécution entre différents aspects.

Si on regarde avec attention certains de ces mécanismes, nous pouvons nous apercevoir qu'ils peuvent entraîner d'énormes retombées s'ils ne sont pas utilisés de façon éclairée. Par exemple, les advices de type around n'ont aucune limite quant à la portée qu'ils peuvent avoir. Puisque le déclenchement de la méthode d'origine est laissé à la discrétion du programmeur, on pourrait à l'aide de ce type d'advice déclencher l'exécution de l'advice au moment de l'appel d'une méthode sans jamais faire appel au code de la méthode d'origine et ainsi remplacer entièrement d'une certaine façon l'exécution d'une méthode par un advice de type around. Évidemment, ce genre de pratiques ne peut être considérée comme une utilisation correcte de la technologie aspect. Le paradigme aspect est là pour prendre en charge les préoccupations transversales et non pour intervenir dans la gestion des responsabilités primaires des objets. Si on s'attarde ensuite aux introductions de structures, elles aussi peuvent

engendrer de mauvaises conceptions lorsque mal utilisées. Les introductions de structures servent fréquemment à caractériser les classes, à l'aide de classes à étendre ou d'interfaces à implémenter qui ne contiennent aucun élément et donc ne demandent aucune modification des classes étendues. Cela peut être fait pour cerner plus facilement ces classes étendues dans la définition des pointcuts. Cependant, il peut en être autrement. Ces introductions de parents peuvent entraîner une modification statique de la classe à étendre soit dans la classe elle-même ou par des introductions publiques à l'aide d'aspects. Dans le cas où les ajouts nécessaires seraient réalisés dans la classe, nous aurions donc des méthodes ou attributs dans une classe qui n'ont de sens que dans le contexte où la classe est modifiée par cet aspect. Nous nous retrouverions donc avec une classe possédant une dépendance vers un aspect ce qui est totalement contraire à la philosophie du paradigme aspect. Seulement, avec ces quelques exemples, nous pouvons voir que les mécanismes aspects peuvent engendrer de très mauvaises structures de systèmes lorsque mal utilisés et mettent en évidence le manque de maturité actuelle de la technologie aspect.

CHAPITRE 3

PRÉSENTATION DE L'APPROCHE

Notre avons développé une technique de test unitaire des aspects. Notre technique se destine aux environnements aspect suivant le modèle d'AspectJ, en particulier, et notre implémentation est réalisée pour ce langage. Elle peut, cependant, être facilement adaptée aux autres implémentations du paradigme. Étant donnée la nature des environnements aspects tels qu'AspectJ où les aspects sont des entités non instanciables et ne pouvant exister hors d'un contexte d'intégration à une classe [Videira01], notre approche vise le test d'un aspect intégré à une classe. Notre technique se base sur l'analyse de l'intégration des aspects et sur leur influence sur le comportement de la classe pour déterminer ce qu'un test unitaire devrait couvrir.

Notre approche supporte la génération de séquences de test à partir des diagrammes d'états-transitions UML. Cependant, le comportement des aspects ne peut être représenté entièrement dans un diagramme d'états-transitions propre à l'aspect. Le comportement dynamique de l'aspect reste tributaire du contexte de chacune des classes auxquelles il est relié. Notre technique consiste à générer, à partir du diagramme d'états-transitions original d'une classe, un nouveau diagramme d'états-transitions étendu représentant le comportement dynamique de la classe incluant les effets (intégration) d'un ou de plusieurs aspects qui lui sont reliés. Le nouveau diagramme illustre à la fois le comportement original de la classe ainsi que l'intégration des aspects. C'est à partir de ces nouveaux diagrammes que les séquences de test seront générées en fonction des critères de test que nous avons développés. Les séquences de test générées tiennent compte donc du comportement de la classe ainsi que des aspects qui lui sont reliés. L'objectif principal de notre technique est de vérifier que le comportement original d'une classe n'est pas altéré suite à l'introduction d'un ou de plusieurs aspects, d'une part, et de s'assurer du respect des spécifications des aspects, d'autre part.

Les diagrammes d'états-transitions UML permettent de modéliser le comportement dynamique individuel des objets d'une classe. Cependant, lorsqu'il s'agit d'introduire des préoccupations transverses de façon modulaire à un diagramme d'états-transitions, il devient difficile de comprendre et de maintenir un tel diagramme. L'article de Mahoney et al. [Mahoney01] décrit bien cette situation. Dans ce contexte, et dans le but de faciliter la représentation de l'introduction des aspects dans le comportement d'une classe, nous considérons à la fois le diagramme d'états-transitions de la classe sous test et le code source des aspects qui lui sont reliés. L'outil se charge d'analyser et de fusionner automatiquement le tout pour générer le diagramme d'états-transitions résultant de l'intégration de l'aspect à la classe. Nous évitons, ainsi, à l'utilisateur la construction et le maintien, qui peut être complexe et fastidieux, des diagrammes d'états-transitions regroupant les préoccupations primaires et transversales.

La figure 5 illustre les principales étapes de la méthodologie que nous proposons. Un seul aspect est considéré dans ce cas. Les éléments de départ, nécessaires à la génération des séquences de test, sont le diagramme d'états-transitions de la classe décrit en XML et le code de l'aspect écrit en AspectJ.

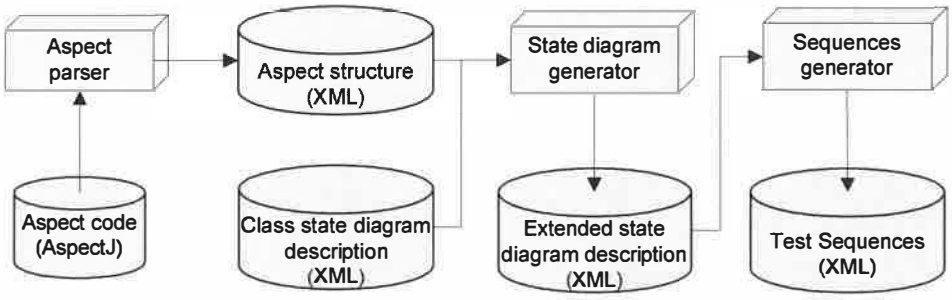

Figure 5 Générateurs de Séquences de Test: Architecture

La première étape de la méthode consiste à analyser le code de l'aspect et en extraire un modèle XML reflétant sa structure. L'étape suivante consiste à analyser conjointement les descriptions XML du diagramme d'états-transitions de la classe et de l'aspect. Seul

le contenu de l'aspect relatif à la classe en question sera considéré. À partir de cette analyse, nous identifions formellement l'influence qu'aura l'aspect sur le diagramme d'états-transitions de cette classe. Le résultat de cette analyse consiste en une nouvelle description XML du diagramme d'états-transitions représentant le diagramme original de la classe étendu par l'aspect. L'étape finale correspond à la génération des séquences de test à partir de cette dernière description XML tenant compte des critères de test que nous avons définis.

La démarche adoptée est incrémentale et itérative. La figure 6 illustre le principe de notre approche globale. Dans le cas où plusieurs aspects sont intégrés à une classe, la démarche consiste à les intégrer un à la fois, en commençant par le plus complexe. Le choix de l'aspect à considérer est basé sur des critères tels que la complexité intrinsèque de l'aspect et son couplage vis-à-vis de la classe en question. Le test d'un bloc aspect-classe se fait de façon incrémentale, en fonction des séquences générées. Les cas de test sont archivés. Dans le cas d'une modification quelconque, apportée soit à un aspect ou à la classe, les cas de test précédents seront réutilisés. L'impact de la modification sur les séquences de test sera identifié formellement. Seules les séquences touchées (directement ou indirectement) seront retestées. Cela évitera, comme c'est le cas de la méthode proposée par D. Xu et al. dans [Xu01], de tout retester comme un seul bloc. La démarche que nous préconisons permet ainsi de réduire l'effort de test. Par ailleurs, et au-delà des séquences de test elles-mêmes, l'intégration incrémentale permettra de mieux se concentrer et identifier les conflits éventuels entre les aspects. La découverte des erreurs éventuelles dans ce cas sera facilitée.

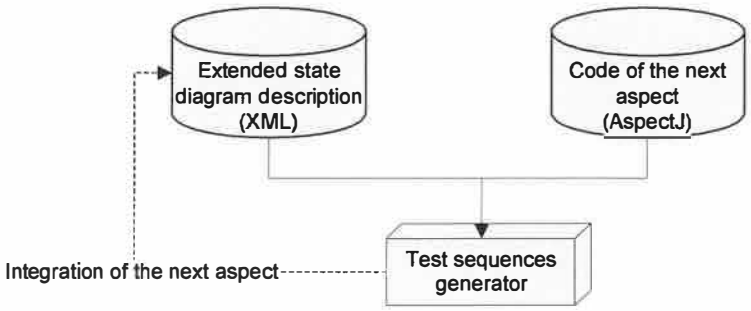

Figure 6 Diagramme des étapes de la méthode

La démarche incrémentale et itérative adoptée nous permet, comme mentionné précédemment, de réduire l'effort de test. Par exemple, et comme illustré par la figure 6, la description XML du diagramme d'états-transitions de la classe sous test peu être remplacée par la description XML de la classe étendue suite à l'introduction d'un aspect. Ainsi, dans une situation où une classe est affectée par plusieurs aspects, nous pouvons tester cette classe de façon incrémentale en incluant progressivement les aspects. Par ailleurs, l'intégration incrémentale des aspects, et grâce à l'analyse formelle des différentes descriptions retenues, nous permet d'identifier formellement le résultat de chaque extension de la classe (intégration d'un aspect) et par la même, mieux orienter le processus de test et identifier les conflits éventuels entre les aspects. Les séquences pouvant conduire à des conflits entre les aspects feront l'objet d'une attention particulière.

3.1 Impacts des aspects sur les diagrammes d'états

Les aspects s'introduisent dans le comportement général des classes pour effectuer les actions nécessaires à la réalisation de leurs objectifs. Ainsi, le comportement de l'ensemble de la classe est sujet à être influencé par un ou des aspects et ces nouveaux comportements doivent être testés. De plus, il faut également tester les interférences possibles qui peuvent se produire entre plusieurs aspects qui se greffent à une même classe. Pour y parvenir, nous passerons par les diagrammes d'états des classes, mais

pour couvrir l'ensemble de ce qui est touché par les aspects nous devons déterminer comment les aspects influencent les classes à travers les modifications qu'ils apportent aux diagrammes d'états des classes. Il existe quatre cas où un aspect peut modifier directement le diagramme d'états d'une classe. Il faut ajouter à cela les cas où les aspects ajoutent de nouvelles zones orthogonales dans les diagrammes d'états sans toucher à la structure d'origine tel que les cas présentés dans l'article [Mahoney01].

3.2 Cas 1 : Advice before et after

Nous présentons le premier cas de modification où une méthode est touchée par l'exécution d'un advice de type before ou after. Ces deux types d'advice sont très semblables dans leur comportement comme mentionné auparavant et ont la possibilité de faire le même type de modification. Habituellement, ces types d'advice ne devraient jamais modifier la structure du diagramme d'états. Ils ne font qu'insérer une section de code avant ou après l'exécution du jointpoint pour réaliser un travail transversal. Nous pouvons voir ceci à travers l'exemple suivant d'une pile. Nous voyons sur la figure un advice de type after sur la méthode Empiler permettant de garder la trace des opérations. L'advice n'a été représenté que sur la méthode Empiler entre les états vide et intermédiaire.

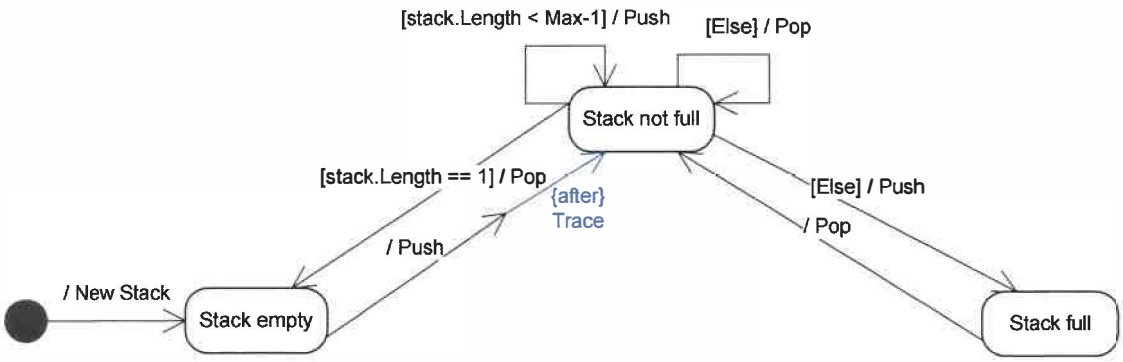

Figure 7 Diagramme d'états avec advice after

Comme on le voit, cette intervention de l'advice ne modifie pas le comportement dynamique de la classe. Cependant, dans certains cas il est possible qu'un de ces advices modifie la valeur d'un invariant de l'état destination et ainsi dirige la classe vers un autre état que celui prévu dans son diagramme. On pourrait penser, par exemple, à un advice de type before qui augmente d'un l'attribut de la classe Pile qui contient le nombre d'éléments dans la pile. Dans les cas où il ne reste qu'un espace disponible dans la pile et que cet advice before augmente d'un l'attribut, la méthode empiler qui s'exécutera par la suite détectera que la pile est pleine alors qu'elle ne l'est pas réellement et lèvera une erreur plutôt que d'empiler la valeur dans le dernier espace disponible. Ces manipulations sont plus des erreurs de conception qu'une utilisation responsable de la technologie aspect, mais il s'agit tout de même d'un impact possible à surveiller.

3.3 Cas 2 : Advice around

Le second cas de modification est causé par les advices de type around et leur particularité de déclencher manuellement l'exécution du code désigné par le jointpoint. Contrairement aux autres types d'advices, ceux de type around n'exécutent pas invariablement le code du jointpoint. Cette caractéristique permet d'importantes manipulations du comportement de la classe d'origine. Ces modifications peuvent aller de la simple insertion de code avant et après l'exécution du jointpoint, à la façon de deux advices before et after, jusqu'au remplacement total du code du jointpoint par celui de l'advice. Nous avons ici, dans cet exemple, le cas le plus simple où un advice around se contente d'exécuter du code avant et après le jointpoint sur la même méthode Empiler entre les états Vide et intermédiaire :

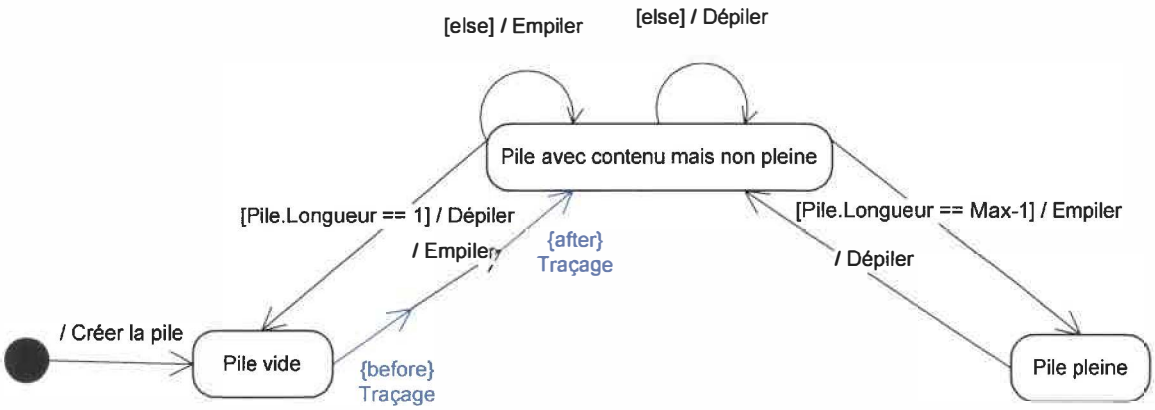

Figure 8 Diagramme d'états avec advice before et after

Donc, dans le cas le plus simple seule une modification des invariants de l'état destination, avant ou après le jointpoint, pourrait modifier le comportement de la classe exactement de la même manière que dans le premier cas. Une des manipulations que l'on pourrait dire plus complexe serait l'exemple où un advice around vérifie si la valeur à empiler est déjà présente dans la pile et si c'est le cas ne pas l'empiler pour ainsi avoir une pile sans doublons. Une manipulation du genre pourrait dans certain cas modifier le comportement dynamique de la classe, par exemple en allant de l'état intermédiaire à intermédiaire plutôt qu'à l'état pleine sur l'appel de la méthode Empiler sur un doublon lorsqu'il ne reste qu'un espace libre dans la pile. Le diagramme nous donnerait ceci :

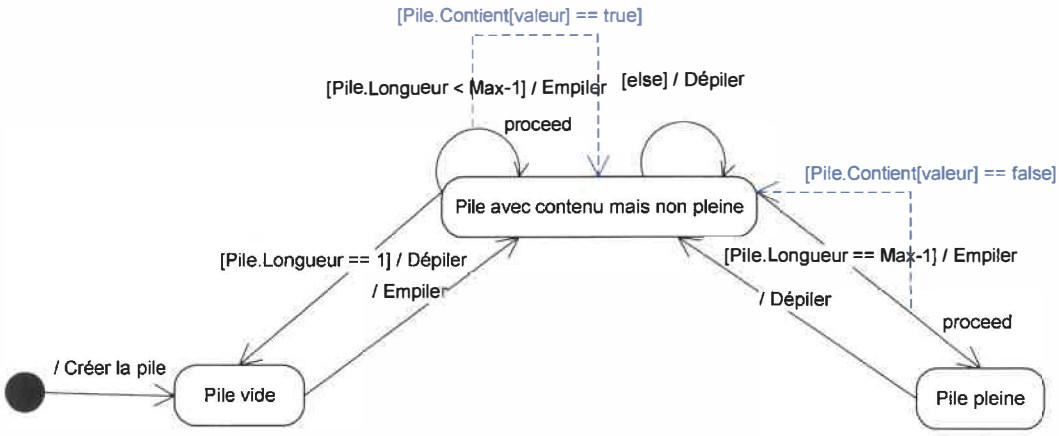

Figure 9 Diagramme d'états avec advice around

Nous voyons que si la valeur n'est pas déjà dans la pile, le contrôle se poursuit normalement peu importe le nombre d'éléments dans la pile mais que dans le cas inverse la classe reste dans l'état Vide puisque la méthode empiler n'a pas été exécutée par l'advice around. Ce genre de modification altère le comportement dynamique de la classe. Il s'agit d'un cas qui peut avoir son utilité mais qui nous montre aussi l'importance de pouvoir tester autant le respect des spécifications de la classe d'origine que des aspects pour éviter une dégradation du travail qui doit être réalisé par la classe. Mais, dans le cas le plus extrême où le jointpoint n'est jamais exécuté par l'advice plus aucune règle n'existe et seule une analyse du code de l'advice pourrait nous dire quel impact aurait l'exécution de cet advice sur l'état de la classe. Dans un cas pareil, on pourrait, sur un appel de la méthode Empiler, réaliser à la place un appel de la méthode Dépiler. Il n'existe aucune limite aux impacts de ce genre. De plus, il s'agit dans ces cas de jeter à la poubelle la classe d'origine et non de la compléter sur les aspects transversaux. Il ne s'agit donc pas d'utilisation correcte de la technologie aspect. Comme nous pouvons le voir, les advices de type around sont des outils excessivement permissifs qu'il est facile d'utiliser à tort et à travers. Quelques règles sur la bonne façon d'utiliser cet outil seraient peut-être nécessaires.

3.4 Cas 3 : Introduction publique

Le troisième cas de modification vient des introductions de méthodes publiques. Les aspects permettent d'ajouter aux classes de nouvelles méthodes qui sont ensuite considérées comme des méthodes de cette classe. Les introductions peuvent être privées ou publiques. Une introduction privée n'est visible que de l'aspect qui déclare cette introduction et n'a donc aucune conséquence sur le diagramme d'états de la classe. Les introductions publiques à l'inverse sont visibles par tous les aspects et classes du système. Cependant, une classe qui ferait appel à une de ces introductions publiques aurait alors une dépendance vers l'aspect où est déclarée cette introduction, ce qui est contraire au principe du paradigme aspect. La classe qui utiliserait ces méthodes publiques, qui font partie d'une préoccupation transversale, se mêlerait alors du travail de cette préoccupation en la déclenchant manuellement, alors que le travail d'une classe est de faire son travail spécifique et c'est aux aspects de gérer complètement les préoccupations transversales. Cependant, une telle méthode publique viendrait s'ajouter à l'interface publique de la classe d'origine et ajouterait donc au moins un lien dans le diagramme d'états de la classe. Nous pouvons voir un exemple ici de l'ajout d'une méthode Vider :

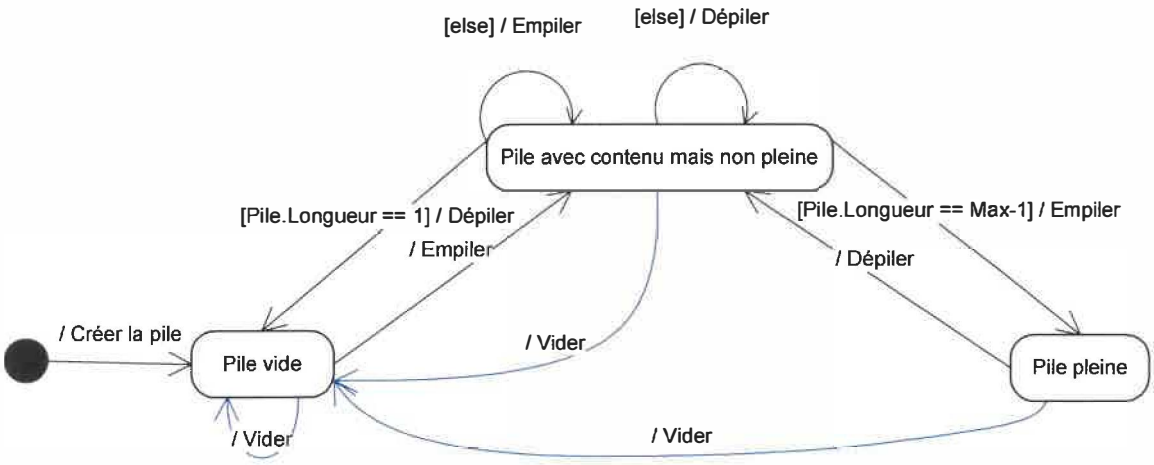

Figure 10 Diagramme d'états avec introduction

3.5 Cas 4 : Introduction de zone orthogonale

Le dernier cas de modification qu'un aspect peut faire sur un diagramme d'états d'une classe est lorsqu'il lui ajoute une nouvelle zone orthogonale dans son diagramme d'états qui représente le comportement dynamique de l'aspect au cours de la vie de la classe. Bien que ce ne soit pas une modification directe, elle doit être prise en compte car le comportement de la classe tout au cours de son cycle peut être influencé par l'état dans lequel se trouve l'aspect qui agira alors différemment selon son état et celui de la classe. En voici un exemple où, pour des raisons de zone de mémoire partagée, la pile en barré avant chaque utilisation et débarrée à la fin de son utilisation :

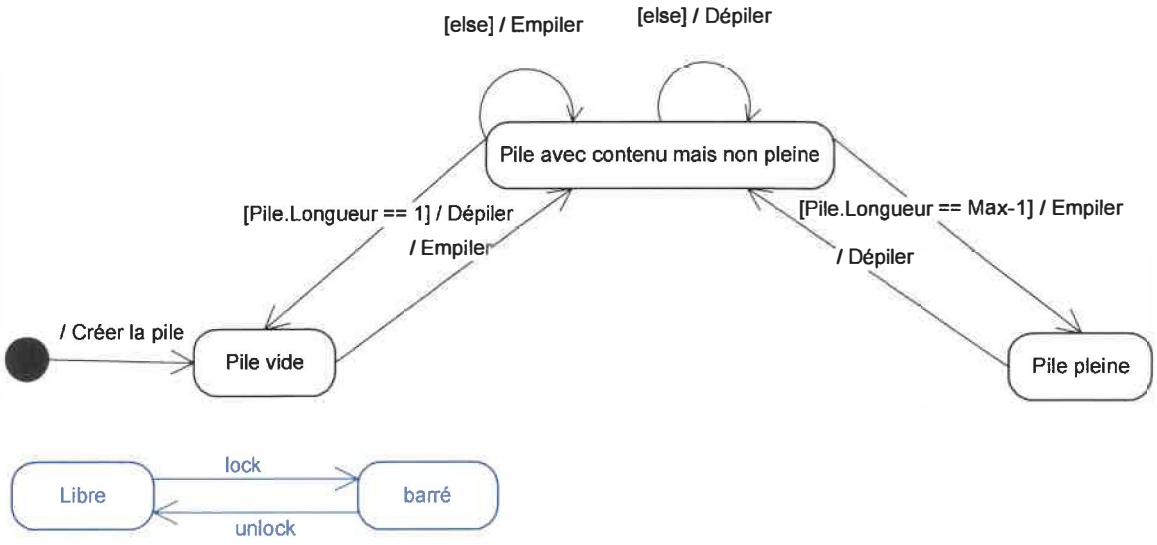

Figure 11 Diagramme d'états avec zone orthogonale ajoutée

Il faut voir qu'aucun mécanisme aspect ne peut permettre directement un tel impact sur un diagramme d'états d'une classe. Il s'agit d'ensemble d'éléments qui au final donne l'équivalent de ce résultat. Dans cet exemple, l'aspect doit premièrement faire une introduction privée d'une variable qui contiendra l'état Libre ou barré (par exemple un booléen). Il devra ensuite introduire deux méthodes publiques (lock et unlock) pour que les utilisateurs de la classe puissent modifier cet état et finalement un advice around sur chaque état qui s'assura que la pile est Libre avant de permettre l'exécution de la méthode appelée. Bien qu'il s'agisse d'une composition d'éléments dont nous avons déjà présenté les effets sur les diagrammes d'états, nous croyons qu'il est important de voir également ce que des compositions peuvent donner car il s'agit de la seule façon d'obtenir un tel impact. Mais, si on regarde de plus près, en fait, il ne s'agit que du cas où des advices around filtrent les exécutions de méthodes selon des conditions sauf que nous utilisons les introductions pour contrôler cette condition. La différence réside donc dans la nature de la condition, est-ce qu'elle est sur un élément de la classe ou un élément introduit par l'aspect.

CHAPITRE 4

CRITÈRES DE TESTS GÉNÉRAUX BASÉS SUR LES GRAPHES D'ÉTATS

Pour ce qui est des critères de génération de séquences de test à partir d'un diagramme d'états UML, l'article [Briand01] présente une étude sur les performances et les coûts de différents critères de génération de séquences de test à partir de diagramme d'états. Il évalue les critères AT (All transitions), All Transition Pairs (ATP), all paths in the statechart Transition Tree (TT) et Full Predicate (FP).

Le critère All Transitions dit que chaque transition doit être testée au moins une fois. Donc, pour ce test on doit vérifier que l'objet passe de l'état de pré transition à celui de post transition pour chaque transition [Offutt01].

Le test Full Predicate est une extension du critère précédent. Il stipule que l'ensemble des cas pour chaque transition doit être testé. Ainsi, lorsqu'il existe une clause sur une transition, toutes les possibilités de cette clause doivent être couvertes par des séquences de test. S'il existe plusieurs clauses sur une même transition, chacune d'entre elles doit être prise indépendamment et toutes les combinaisons possibles des différentes clauses doivent être couvertes par les séquences de test. Ainsi, l'ensemble des comportements de chaque transition est testé [Offutt01].

Alors que les deux premiers critères étaient en fait le même, mais à différents niveaux, les deux derniers critères eux sont indépendants. Le critère All Transition Pairs stipule que chaque paire de transitions adjacentes doit être testée en séquence [Offutt01].

Le dernier critère All paths in the statechart Transition Tree est aussi appelé Complete Sequence dans certains articles. Ce critère se base sur l'idée que le fonctionnement individuel de l'ensemble des parties hors du contexte d'une exécution complète ne

garantit pas le fonctionnement correct de ces parties une fois regroupées ensemble. Alors, ce critère dit que nous devons tester le système avec des séquences de test qui couvrent le diagramme d'états de bout en bout. Cependant, dans bien des cas le nombre de possibilités est presque infini et il n'est pas possible de couvrir l'ensemble des possibilités. Il faut alors une certaine connaissance du domaine pour proposer les séquences de test parmi les possibilités existantes qui se rapprochent le plus de l'utilisation réelle du système et ainsi tester les cas les plus probables [Offutt01].

À la lecture de la conclusion de l'article [Briand01], il est clair qu'AT n'est pas coûteux mais n'offre pas une très bonne détection des erreurs. À son inverse, ATP offre une détection presque parfaite mais à un coût très élevé. Le critère TT semble le meilleur compromis entre ces deux extrêmes lorsque les diagrammes d'état n'ont pas de conditions de garde (guard conditions). Quant au critère FP, il semble être un compromis acceptable entre les critères AT et ATP dans certaines situations où des conditions de garde sont présentes. Ces critères nous permettent de générer des séquences de test pour les classes, mais une fois qu'une de ces classes a été étendue par un ou des aspects aucun des critères présentés ne peut nous dire qu'est-ce qui doit être retesté. Notre but étant de ne pas tout retester, mais uniquement ce qui doit l'être. C'est là que les critères que nous voulons développer interviennent. Ils permettront de savoir ce qui doit être considéré comme non testé (à retester) et générer les séquences de test pour couvrir ces zones.

CHAPITRE 5

CRITÈRES DE TEST DÉVELOPPÉS

Dans cette section, nous présenterons les nouveaux critères de génération de séquences de test pour couvrir les nouvelles dimensions introduites par le paradigme aspect et discutées auparavant. Les critères qui ont été présentés dans la dernière section permettent de générer des séquences de test à partir de diagrammes d'états de classes. Mais, comme mentionné précédemment, les aspects ont différentes formes d'influence sur le comportement et le diagramme d'états de ces classes. Les critères existants ne permettent pas de générer les tests nécessaires pour couvrir les nouveaux comportements introduits par ces aspects puisqu'ils ne prennent pas compte leur existence et donc encore moins leur influence. Pour couvrir tous ces éléments et nous assurer du respect des spécifications après l'insertion des aspects au code objet, nous devons créer de nouveaux critères qui couvrent les possibilités de tous les mécanismes du paradigme aspect. Pour cela, nous avons développé quatre nouveaux critères pour couvrir ces mécanismes : (1) Les méthodes touchées par un jointpoint doivent être considérées comme de nouvelles méthodes non testées; (2) Toutes les possibilités d'exécution des advices doivent être couvertes; (3) Les modifications statiques de la structure du diagramme d'états doivent être couvertes par les séquences de parcours de graphe; (4) L'ensemble des paires de transitions classe - aspect doivent être couvertes dans le cadre d'un diagramme d'états composé.

Ces critères permettent, pour le moment, de couvrir les possibilités de l'introduction d'un aspect à une classe. Mais plus tard, dans le développement de la méthode, ces critères pourront être généralisés aux cas où plusieurs aspects touchent une classe. À ce moment, de nouveaux critères, pour couvrir cette nouvelle dimension, seront développés. Les autres méthodes des articles présentés précédemment ne présentent pas de critères décrivant ce qui doit être testé pour couvrir les particularités du paradigme

aspect. L'article [Xu02] utilise des critères se basant sur une nouvelle définition des associations definition-use. Cependant, cette façon de faire ne permet pas de couvrir l'ensemble des situations. L'article [Alexander01] nous présente un modèle de fautes nous guidant vers les points qui sont à surveiller pour s'assurer de capturer l'ensemble des erreurs. Mais, il s'agit d'un modèle très généraliste qui ne nous indique pas ce qui doit être testé selon les situations en fonction des modifications qu'apportent les aspects au comportement du système. Nos critères de test tentent de définir les tests nécessaires selon les modifications du comportement du système dues aux aspects en se basant sur les altérations possibles des diagrammes d'états des classes. De cette façon, nous voulons mieux cerner les tests que nécessite chaque situation et ainsi couvrir les erreurs possibles présentées dans le modèle de fautes de l'article [Alexander01].

5.1 Les méthodes influencées par un jointpoint doivent être retestées

Lorsqu'une méthode est affectée lors de son exécution par l'exécution d'un advice before, after ou around, son comportement général est alors modifié. Il peut être altéré. Pour ce qui est des advices de type around, ce critère s'applique lorsque ces advices ne causent aucune modification de la structure du diagramme d'états-transitions de la classe. Après l'intégration du code de l'aspect à la méthode en question, nous nous retrouvons avec une nouvelle méthode étendue dont le comportement doit respecter les spécifications de la méthode d'origine tout comme celles de l'advice. On peut voir sur ce diagramme un exemple d'advice after :

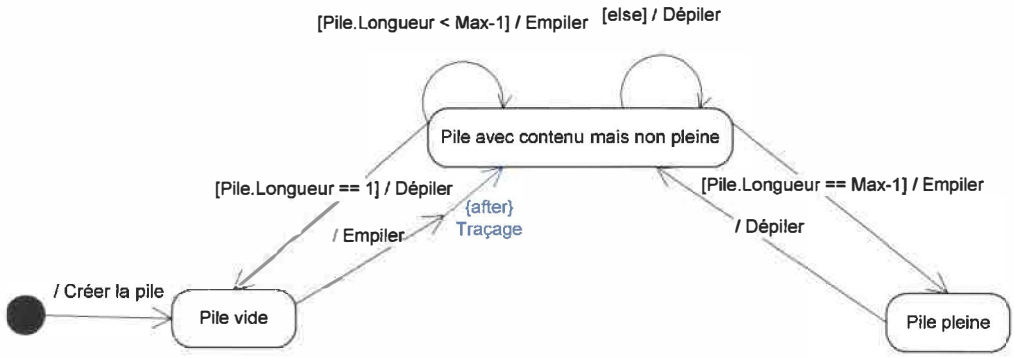

Figure 12 Diagramme d'états avec advice after

Cet exemple illustre le cas de la méthode Empiler de la classe Pile étendue par le code d'un advice de traçage qui est exécuté juste après celui de la méthode. Dans cet exemple, l'advice n'a été noté que sur la transition Empiler entre les états Pile vide et Pile avec contenu à la seule fin d'exemple. Donc, nous avons une nouvelle interaction entre l'advice de traçage et la méthode Empiler. Cette interaction peut altérer le bon fonctionnement de la méthode Empiler. Il faut donc tester cette nouvelle méthode étendue dans les séquences de test même si la méthode d'origine a déjà été testée. Il faudra alors tester ce type de séquences selon les critères de test classiques et vérifier le respect des spécifications de l'advice et non uniquement de la méthode. Ce critère se veut surtout un marqueur qui permet d'identifier les méthodes affectées par les aspects et non de dire comment il faut les tester. Cependant, une fois que ces méthodes sont marquées et les nouvelles séquences identifiées, il est relativement facile de générer à partir des critères classiques les séquences de test nécessaires pour couvrir ces méthodes qui ont besoin d'être testées en particulier et non l'ensemble des transitions du diagramme d'états. Il faudra ajouter à ces critères classiques la vérification des spécifications des aspects en plus de ceux de la classe.

5.2 Toutes les possibilités d'exécution des advices doivent être couvertes.

Les pointcuts ne se déclenchent pas toujours automatiquement lorsque leurs jointpoints sont rencontrés, les exécutions des advices peuvent être conditionnelles. Donc, lorsque l'exécution d'un advice est conditionnelle, son comportement peut varier selon la condition et le contexte. Voici un exemple dans la figure 4 d'un advice after, exécuté après la méthode Empiler partant de l'état Pile vide, dont l'exécution dépend de la condition [!Pile.Contient(NewElement)]. Cependant, seule l'exécution de l'advice est conditionnelle. L'exécution de la méthode originale est quant à elle assurée.

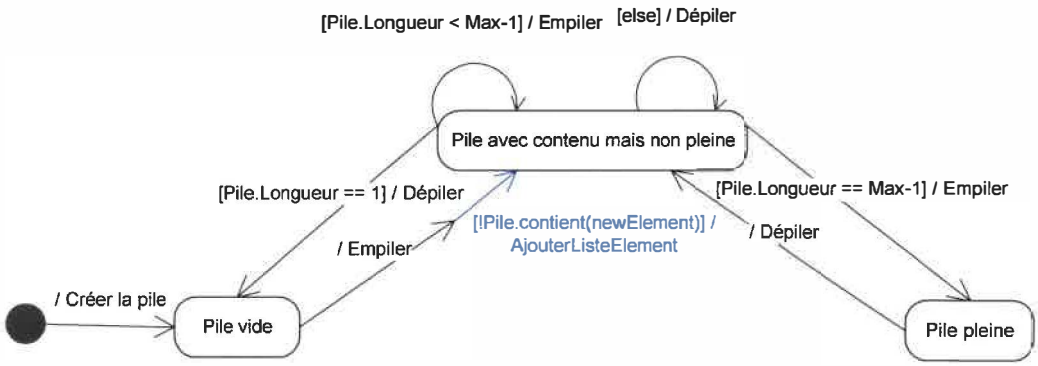

Figure 13 Diagramme d'états avec advice conditionnel

Dans cet exemple, si l'élément ajouté à la pile n'y était pas déjà, l'advice l'ajoute à la liste des éléments présents dans la pile. Encore une fois, l'advice n'est représenté que sur une seule transition uniquement à des fins d'exemple. Ces conditions peuvent être booléennes, dépendre de la levée d'une erreur ou d'autre nature. Puisque chacune de ces possibilités peut aboutir à un résultat différent, on doit couvrir l'ensemble de ces cas par des séquences de parcours de graphe. Nous élaborons alors le critère selon lequel les séquences de parcours de graphe devront couvrir l'ensemble des possibilités d'exécution des advices pour les méthodes étendues. Ce critère est une adaptation du critère Full Predicate aux particularités d'exécution des advices.

5.3 Les modifications statiques de la structure du diagramme d'états doivent être couvertes par les séquences de parcours de graphe

Comme mentionné précédemment, il existe différents mécanismes du paradigme aspect qui peuvent avoir une influence sur la structure statique des classes ainsi que leur diagramme d'états-transitions. Les introductions de méthodes publiques, les introductions de structures ou encore les advices de type around dans certains cas peuvent modifier cela.

Les introductions de méthodes publiques ajoutent de nouvelles transitions dans les diagrammes d'états des classes. Les introductions de structures, que ce soit une classe à étendre ou une interface à implémenter, peuvent demander des modifications statiques à la classe ainsi étendue pour respecter les nouvelles règles introduites par ces structures. Ces modifications passent souvent par la création ou l'introduction de nouvelles méthodes publiques. Les advices around, quant à eux, peuvent détourner complètement le comportement d'une classe puisqu'ils ont le contrôle absolu sur l'exécution de la méthode d'origine. Nous ne discuterons pas ici du bien-fondé de l'utilisation de ces constructions d'une façon ou d'une autre. Mais, seulement de la possibilité qu'ils ont de modifier la structure des diagrammes d'états-transitions et de leurs conséquences en termes de séquences de test nécessaires. Nous avons ici, par exemple dans la figure 5, un diagramme d'états-transitions avant et après l'effet d'un advice de type around qui exécute la méthode d'origine dans le cas de la condition où la pile ne contient pas déjà l'élément à empiler, de plus on peut voir l'introduction de la méthode publique Vider :

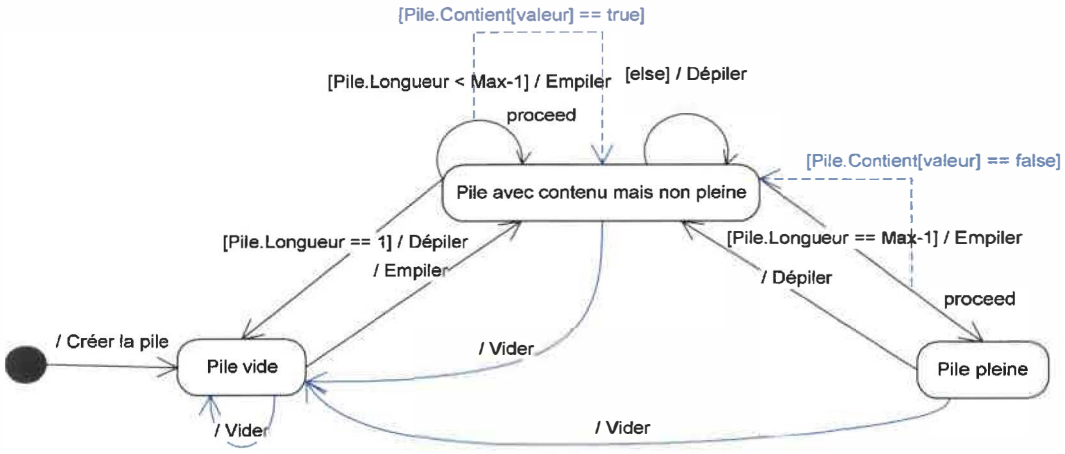

Figure 14 Diagramme d'états avec advice around et introduction

Donc, on peut voir que certains mécanismes, lorsqu'utilisés d'une certaine façon, peuvent entraîner des modifications statiques aux diagrammes d'états. Ces modifications qui comprennent de nouvelles transitions, états ou conditions doivent être considérées comme non testées et les séquences de parcours de graphe doivent être générées pour couvrir ces nouveaux chemins de la même façon que les chemins du diagramme d'états d'origine. Ainsi, de nouvelles séquences de test devront être générées en respect des critères classiques choisis pour couvrir ces nouveaux éléments du diagramme en particulier, ce qui comprend toutes les séquences où l'un de ces nouveaux éléments est présent.

5.4 L'ensemble des paires de transitions classe - aspect doit être couvert dans le cadre d'un diagramme d'états composé

Dans certains cas, il est possible qu'un aspect qui se greffe à une classe ne modifie pas la structure du diagramme d'états de cette classe, mais qu'il a son propre sous diagramme d'états dont les transitions correspondent à certaines transitions du diagramme d'états de la classe. Par exemple, nous avons ici une figure extraite de l'article [Mahoney01] qui représente une classe tampon (buffer) avec deux aspects qui viennent s'y greffer.

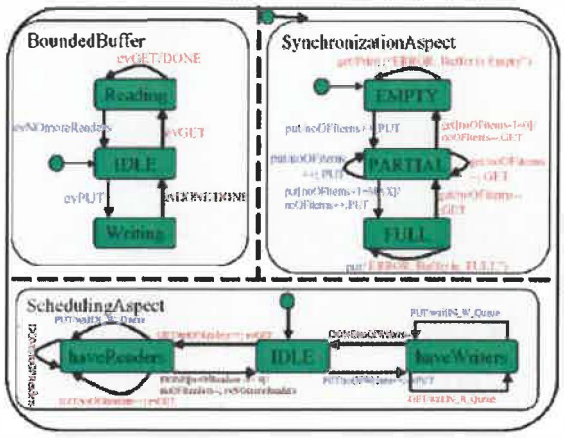

Figure 15 Diagramme d'états à plusieurs régions orthogonales extraites de [Mahoney01]

Chacun de ces aspects, de synchronisation et de gestion d'accès (mode lecture-écriture), ont leur propre sous diagramme d'états où chacune de leurs transitions correspond à une transition du diagramme de la classe. En pratique, ces liens entre les transitions de la classe et des aspects se font à travers les pointcuts et advices. Nous savons que le comportement dynamique de cette classe respecte les spécifications puisqu'elle a déjà été testée unitairement grâce aux méthodes orientées objet. Mais, rien ne nous garantit du bon fonctionnement d'un aspect ainsi greffé dans ce contexte particulier pas plus que de la non-détérioration de ce qui était déjà testé dans la classe. Il faut donc être en mesure de tester le comportement de l'aspect dans ce contexte ainsi que du maintien du bon fonctionnement de la classe. Pour cela, nous devons nous assurer du bon fonctionnement des deux entités pour toutes les paires de transitions classe - aspect. Par là, nous voulons dire que si on prend l'exemple de l'article [Mahoney01] mais uniquement avec la classe et l'aspect de synchronisation, il faudra alors couvrir la transition « put » de la classe qui fait passer l'état d'« Idle » à « Writing » pour les quatre cas de transition d'état de l'aspect qui lui sont associé, soit les transitions qui font passer l'aspect de Empty->Partial, Partial->Partial, Partial->Full et Full->Full. Chacune de ces possibilités doit être couverte, car le comportement de l'advice peut varier en fonction de l'état de l'aspect et avoir un effet différent d'une fois à l'autre sur la méthode étendue. Par exemple, toujours avec Put, la transition Empty à Partial ne lève

aucune exception alors que Full à Full lève l'exception « Buffer is full ». Si nous ne testons que Empty à Patial, il est possible que la méthode originale ne soit pas affectée et respecte toujours ses spécifications. Mais, que lors de la levée de l'exception à la transition de Full à Full, il en soit autrement puisque le déroulement de l'advice est différent car dans un autre contexte. Il faut donc générer des séquences de test qui permettront de couvrir l'ensemble des paires de transitions classe-aspect dans le cas d'aspects ayant leur propre sous diagramme d'états. L'article [Mahoney01], ne se voulant pas présenter une méthode de test mais bien un framework pour la création et l'entretien de ce type de diagramme, ne propose aucun critère de test quel qu'il soit.

CHAPITRE 6

GÉNÉRATION DES SÉQUENCES DE TEST

Lors du test unitaire d'une classe donnée, il existe comme mentionné précédemment différentes sources d'erreurs possibles. Ces erreurs peuvent provenir soit du code objet de la classe, soit du code aspect, soit de l'intégration du code aspect au code de la classe ou bien des conflits entre les aspects dans le cas de l'intégration de plusieurs aspects. De nombreuses techniques pour le test unitaire des classes existent déjà [Briand01, Offutt01, Offutt02] et ce n'est pas vers cette source d'erreurs que notre technique s'oriente. Notre technique suppose que le code de la classe est déjà testé. Concernant les erreurs provenant du code aspect, comme mentionné précédemment, le code aspect est très dépendant du contexte auquel il est lié et un test générique ne peut couvrir l'ensemble des contextes possibles. Notre approche est incrémentale et consiste à intégrer les aspects à la classe, un à la fois pour mieux orienter le processus de test.

Les techniques classiques de test unitaire des classes basées sur les diagrammes d'états-transitions consistent essentiellement à transformer le diagramme en un arbre et générer à partir d'une analyse de l'arbre l'ensemble des séquences de test. Nous adoptons, dans notre approche, le même principe. Nous mettons l'accent sur l'identification des séquences originales affectées par l'intégration d'un ou de plusieurs aspects. Nous présentons, dans ce qui suit, les principales étapes de notre technique de génération de séquences de test. Elles seront illustrées à l'aide des exemples donnés par les tableaux 1 et 2 ainsi que les figures 16 et 17. Les tableaux 1 et 2 présentent le code de la classe Stack et le code de l'aspect StackAspect considéré. Cet aspect contient trois advices, ses advices before et after permettent de faire le traçage des entrées et des sorties de chaque méthode publique de la classe Pile alors que son advice around empêche que des doublons se retrouvent dans la pile. L'aspect contient également une introduction privée qui lui permet de savoir si une pile contient un élément en particulier. La figure 16

présente le diagramme d'états-transitions original de la classe Pile et la figure 17 présente le diagramme d'états-transitions étendu de la classe Pile après intégration de l'aspect AspectPile.

```
public class Stack {
        private Node _firstNode;

        public Stack() {
                this._firstNode = null;
        }

        public Node getFirstNode() {
                return this._firstNode;
        }

        public void setFirstNode(Node node) {
                this._firstNode = node;
        }

        public void Push(Node node) {
                node.setNextNode(this._firstNode);
                this._firstNode = node;
        }

        public Node Pop() {
                Node popNode = this._firstNode;
                this._firstNode = popNode.getNextNode();
                return popNode;
        }

}
```

Tableau 1 Code de la classe Stack

```
public aspect StackAspect {

	private boolean Stack.Content(Node node) {
		Node currentNode = this.getFirstNode();
		while(currentNode != null)
		{
			if (currentNode.equals(node))
				return true;
			currentNode = currentNode.getNextNode();
		}
		return false;
	}

	pointcut StackPublicMethod() : call( public * *(..)) && target(Stack);

	before () : StackPublicMethod() {
		System.out.println("Enter in method : " + thisJoinPoint.getSignature().toString());
	}

	void around ( Node node, Stack stack ) :
		call(public void Stack.Push(Node))
		&& args(node) && target(stack) {
			if ( !stack.Content(node) )
				proceed(node, stack);
		}

	after () : StackPublicMethod() {
		System.out.println("Exit of method : " + thisJoinPoint.getSignature().toString());
	}

}
```

Tableau 2 Code de l'aspect StackAspect

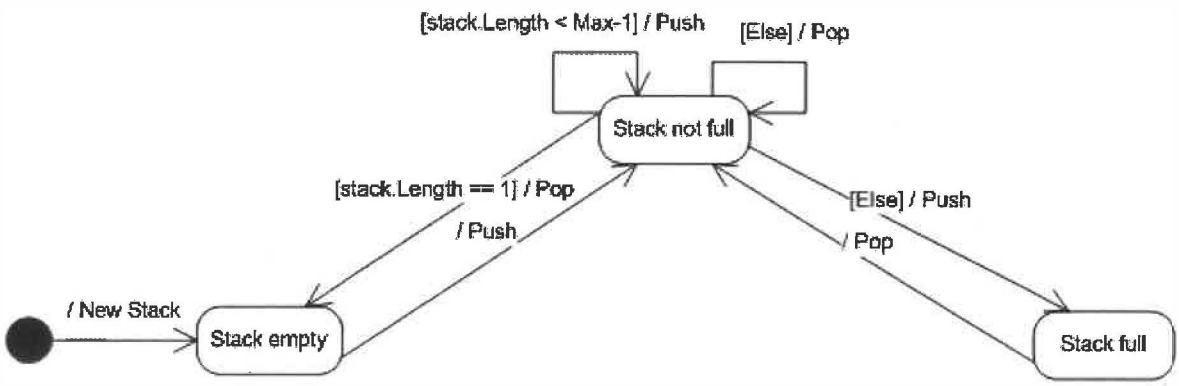

Figure 16 Diagramme d'états de Stack

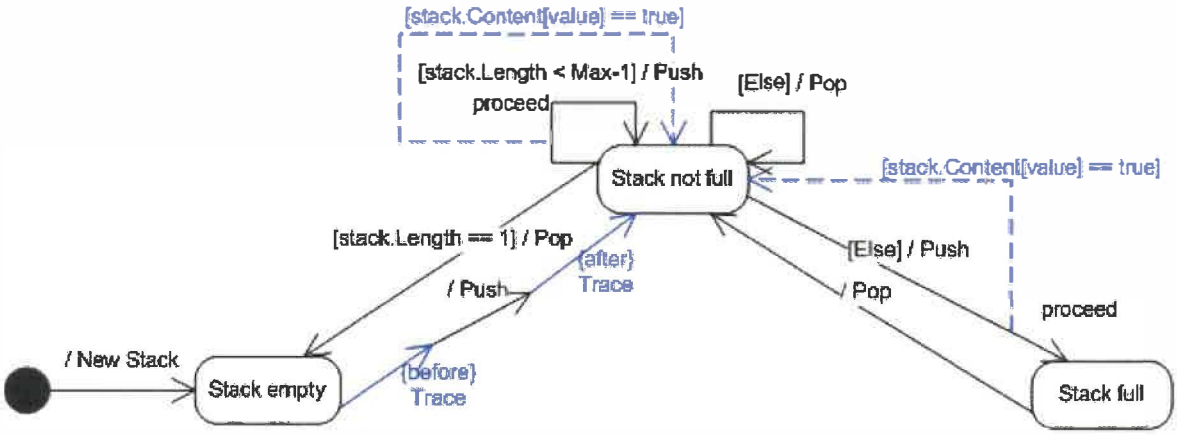

Figure 17 Diagramme d'états Stack avec l'aspect StackAspect

La première étape du processus de test consiste à tester la classe sans les aspects. Cette démarche permet de réduire la complexité du test et d'éliminer dans un premier temps les erreurs relatives au code de la classe. La seconde étape consiste à intégrer de façon incrémentale les aspects à la classe. Il s'agit, dans cette étape, d'identifier les méthodes de la classe qui sont affectées par l'intégration de l'aspect et de déterminer par conséquent les transitions affectées du diagramme d'états-transitions de la classe. Au niveau du diagramme étendu de la classe Pile donné par la figure 9, nous pouvons remarquer que des conditions d'exécution sur la méthode Push ont été ajoutées par l'advice around de l'aspect. Les advices before et after quant à eux s'exécutent avant et après les méthodes Push et Pop de la classe. Dans le but d'éviter de charger le

diagramme et de faciliter sa lecture, nous avons délibérément omis de mettre toutes les modifications apportées aux transitions Push et Pop. La troisième étape consiste à construire l'arbre de parcours des chemins du diagramme (donné par la figure 10) et d'identifier les éléments à tester.

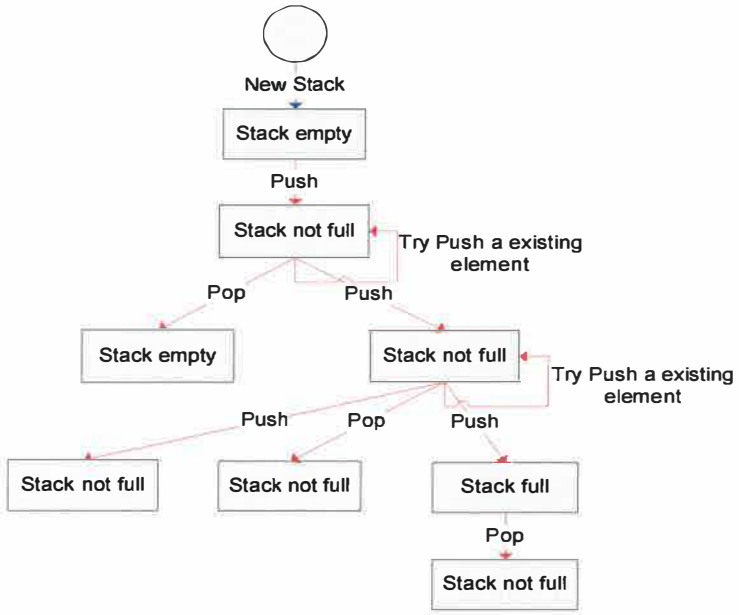

Figure 18 Arbre de parcours du diagramme

Les méthodes qui ont été affectées par l'aspect ainsi que celles introduites sont représentées en rouge. La méthode Contient n'est pas représentée dans l'arbre car elle ne modifie par le comportement de la classe. Elle devrait tout de même être testée pour les trois états possibles. Ces méthodes doivent donc être considérées comme non testées. Nous pouvons ainsi, grâce à la méthode de parcours d'arbre classique All Transitions Path (ATP) [Kandé01], générer l'ensemble des séquences de test concernant les modules affectés par l'aspect AspectPile. Les séquences possibles sont :

- New Stack, Push, Pop
- New Stack, Push, Try Push a existing element, Pop
- New Stack, Push, Push, Push until stack is full
- Etc.

La technique que nous proposons est itérative et incrémentale. Les aspects sont intégrés un à la fois en commençant par le plus complexe. Elle permet à la fois de minimiser l'effort de test et d'orienter le processus de test en cas d'erreur contrairement à certaines approches [Xu01]. Dans l'approche proposée par Xu et al. [Xu01], l'ensemble (classe et aspects) est traité comme un seul bloc. Il est difficile, dans ce contexte, de situer les sources des différentes erreurs. Elles peuvent provenir de n'importe qu'elle entité ou interaction entre ces entités. En intégrant de façon incrémentale les aspects, nous réduisons la complexité du test et nous facilitons la mise au point en cas d'erreur. En cas de défaillance, nous pouvons en effet cibler de manière précise l'origine de l'erreur (réduire le champ d'investigation). Les fautes reliées aux interactions entre les aspects et les classes sont facilement identifiables. Les séquences sont générées en effectuant toutes les combinaisons possibles dans le cas d'une intégration multi-aspects afin de détecter les erreurs relatives au comportement aléatoire possible dans ce cas. Notre stratégie se concentre principalement sur la détection de fautes relatives au processus de greffe des aspects. Notre approche permet de tester la classe avec chacun de ses aspects un à la fois et de s'assurer, au fur et à mesure, du bon fonctionnement de l'intégration de chaque aspect avec la classe.

Les principales étapes de la méthode adoptée sont décrites par l'algorithme suivant :

1. Génération des séquences de test unitaire de la classe.
2. Test unitaire de la classe.
3. Intégration des aspects: Tant qu'il y a des aspects non intégrés
4. Intégration d'un aspect.
    a. Génération du diagramme d'états-transitions étendu de la classe.
    b. Identification des transitions affectées par l'intégration de l'aspect..
    c. Construction de l'arbre correspondant au diagramme étendu
    d. Génération des séquences de test affectées par l'aspect.
    e. Test de la classe avec l'aspect intégré.
    a. Si aucun problème, retour à l'étape 4.
5. Fin

CHAPITRE 7

OUTIL

Ce chapitre présente l'outil que nous avons développé pour mettre en pratique notre approche. L'outil permet de générer automatiquement des classes de test unitaires pour une classe affectée par un ou plusieurs aspects. L'outil génère, dans ces classes de test, les séquences de test qui doivent être testées pour couvrir les critères de test que nous avons présentés. Les classes de test ainsi que la partie de l'outil qui exécute ces tests et produit les résultats sont une extension du framework de test unitaire Java JUnit.

7.1 JUnit

JUnit est un outil supportant les tests unitaires des classes des programmes Java. Nous donnons, dans cette section, un bref aperçu sur l'outil et son fonctionnement de base. Il constitue en fait le noyau de l'environnement que nous avons développé pour supporter notre approche. JUnit contient essentiellement trois composants : la classe Assert, l'interface Test et les Runners. La figure 19 présente la structure générale de JUnit avec les principaux éléments nécessaires à la création et l'exécution d'un test JUnit.

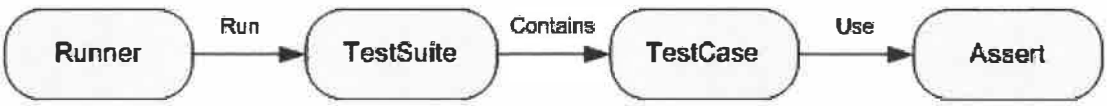

Figure 19 JUnit general structure

La classe Assert contient l'ensemble des méthodes de vérification des valeurs attendues à des points clés dans les méthodes de test. C'est à cette classe que l'on passe la valeur attendue, la valeur réelle lors de l'exécution du test et le message pour le cas où la valeur réelle différerait de la valeur attendue.

TestCase est la classe de JUnit représentant un test unitaire. Tous les tests unitaires de JUnit sont des classes qui dérivent de TestCase. Cette classe contient également toute la mécanique de réflexion pour réaliser les appels aux méthodes de test. Les TestSuite regroupent un ou plusieurs TestCase. C'est cette classe qui définit quelles sont les méthodes des TestCases qu'elle contient qui sont des méthodes de test unitaire, par exemple l'implémentation de JUnit définit que ce sont les méthodes dont le nom commence par le préfix « test » qui doivent être exécutées lors du test.

Finalement, les runners sont des classes qui exécutent les instances de TestCase et/ou TestSuite et permettent d'avoir accès au résultat. Le runners le plus simple de JUnit est TextRunner qui retourne les résultats des tests sous forme texte dans une console. Le plug-in JUnit de l'IDE Eclipse est également un runners mais qui donne son résultat dans un plug-in viewer d'Eclipse.

Pour AJUnit la classe Assert ainsi que tous les runners JUnit peuvent être utilisés. Une spécialisation de la classe TestSuite nous permet d'exécuter des séquences de test au lieu de test isolé. Les classes de test proprement dites de AJUnit sont des extensions de la classe TestSequenceCase qui est elle-même une extension de la classe TestCase de JUnit. Ainsi, toutes les classes de test d'AJUnit sont également des tests unitaires JUnit.

7.2 Structure de l'outil

L'outil que nous avons développé se compose de deux modules principaux. Le premier module est l'analyseur. Ce module fait l'analyse du diagramme d'états de la classe cible ainsi que du code de l'aspect que nous intégrons. Ce module produit une structure contenant toute l'information de chaque advice et pointcut sur chacune des méthodes de la classe analysée. Le second module, celui de génération de tests unitaires, récupère cette structure pour une seconde passe d'analyse. Cette seconde passe consiste à faire un

parallèle entre les différents liens entre la classe et le ou les aspects ressortis par le premier module et les critères de test que nous avons développés pour définir quels sont les tests qui doivent être réalisés pour couvrir l'impact de ou des aspects sur la classe. Une fois les tests établis, le module génère une classe de test contenant les stubs des tests unitaires et des séquences de test qui auront été établies.

7.2.1 Module d'analyse

Le premier module fait une analyse de l'influence de l'aspect sur la classe pour ensuite, à partir de ce portrait, déterminer quelles sont les séquences qui doivent être testées. Ce procédé se découpe lui-même en plusieurs sous modules. On peut voir sur le diagramme ci-dessous les étapes du processus contenues dans le module d'analyse.

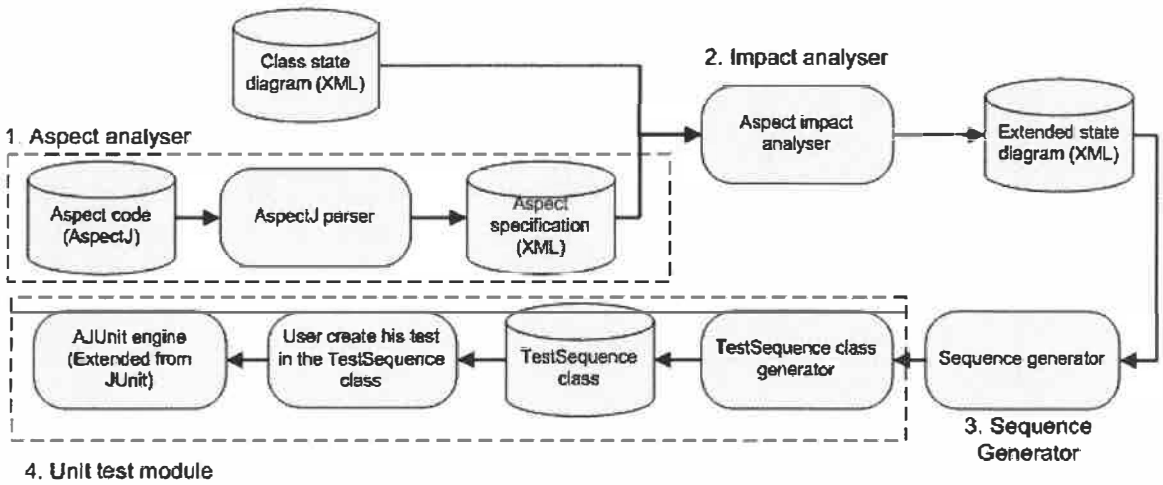

Figure 20 Diagramme du module d'analyse

7.2.1.1 Analyseur de l'aspect

Ce premier sous module (AspectJ parser dans le diagramme ci-haut) prend en entrée le code d'un aspect AspectJ et en fait l'analyse du code pour en faire extraire une structure contenant l'information quant à comment cet aspect se greffe à une classe. C'est-à-dire,

quels sont ses advices, de quel type sont-ils, selon quels pointcuts ces advices s'introduisent dans le flot de contrôle des classes et quels sont les descriptions de ces pointcuts pour pouvoir définir si une certaine méthode d'une certaine classe est touchée par ces advices. Toutes ces informations peuvent être gardées sous la forme d'un document XML. Il s'agit d'un condensé des règles d'intrusions de l'aspect dans le flot de contrôle des classes.

7.2.1.2 Analyseur d'impact

Ce sous module reprend la « carte » de l'aspect générée par le module précédent sous la forme d'un fichier xml ainsi que le fichier xml du diagramme d'états de la classe. Le module fait ensuite l'analyse comparative du diagramme d'états de la classe et du modèle de l'aspect. Cette analyse se base sur la correspondance entre les signatures des méthodes de la classe et des pointcuts de l'aspect. Cette analyse permet de relever les sections de la classe qui sont touchées par l'aspect et par quel advice. Ainsi, chaque méthode qui est visée par une des structures de l'aspect est marquée comme étant affectée par cet aspect en particulier. Ce module produit donc un nouveau diagramme d'états, celui-là étendu, car il contient en plus toute l'information sur les aspects qui touchent chaque méthode. C'est à partir de ce diagramme que le prochain module générera les séquences de test. C'est également à cette étape que le test de plusieurs aspects sur une même classe de façon incrémentale se joue. En effet, ce module peut prendre n'importe quel diagramme d'états en entrée, ce qui inclut des diagrammes d'états contenant déjà l'information de l'impact d'un autre aspect sur cette même classe, l'information de l'impact de l'aspect courant y sera simplement ajoutée et le nouveau diagramme d'états étendu contiendra l'information de l'impact des deux aspects sur cette classe.

7.2.1.3 Générateur de séquences

Une fois que le diagramme d'états étendu comprend l'information quant à l'impact du ou des aspects sur la classe, ce sous module en extrait les séquences de tests qui doivent être couvertes selon les critères que nous avons développés. Une fois ces séquences identifiées, nous générons une classe de test contenant les stubs de test et de séquences de test. La première étape est de créer une méthode de test pour chacune des méthodes faisant partie d'une séquence qui devra être testée. Ensuite, une méthode de test est générée pour chaque séquence et des appels aux méthodes de test correspondantes aux méthodes de cette séquence sont générées pour recréer la séquence tel qu'elle doit être testée selon nos critères. La classe générée dérive de notre classe SequenceTestCase qui elle-même dévire de TestCase de JUnit.

7.2.2 Module de test unitaire

Le composant « Aspect impact analyser » comme présenté dans la dernière 5.1.2 permet à l'utilisateur de générer une classe de test avec les stubs des méthodes et des séquences devant être testées en fonction de l'impact de l'aspect sur la classe. Le développeur peut ainsi y inscrire les tests qu'il désire voir son système réussir en sachant que ces tests couvrent l'ensemble des impacts et comportements possibles de l'aspect sur la classe. Une fois cela fait, ce second module permet de vérifier les tests de la même façon que JUnit vérifie les tests unitaires orienté object. Ce module est une extension de JUnit. Cette extension consiste en la création de deux classes dérivées respectivement de Test et TestSuite. La classe Assert et aucun runners spécifique n'est nécessaire. La classe dérivée de TestCase n'est qu'une dérivation fortement typée. La classe dérivée de TestSuite, quant à elle, remplace la notion de méthode de test commençant par le préfix « test » par la notion de méthodes de séquences de test identifiée par le préfix « sequence ». Cette modification permet d'éviter toute collision avec JUnit. Il est

toujours possible de faire des tests unitaires JUnit dans le même TestCase. Puisque ces modules découlent directement de JUnit, chacun de nos TestCase est également un TestCase JUnit valide. Donc, ces classes de test peuvent toujours être exécutées sous JUnit et fonctionner en parallèle avec nos tests et notre outil sans le moindre problème. De plus, l'héritage laissé par JUnit s'étend jusqu'aux runner qui peuvent être étendus avec un minimum d'effort pour supporter les tests de notre outil. Il s'agit de la dernière étape du processus de test comme le montre le diagramme.

On peut voir ici le diagramme de classe qui montre la relation entre JUnit et le module de test unitaire.

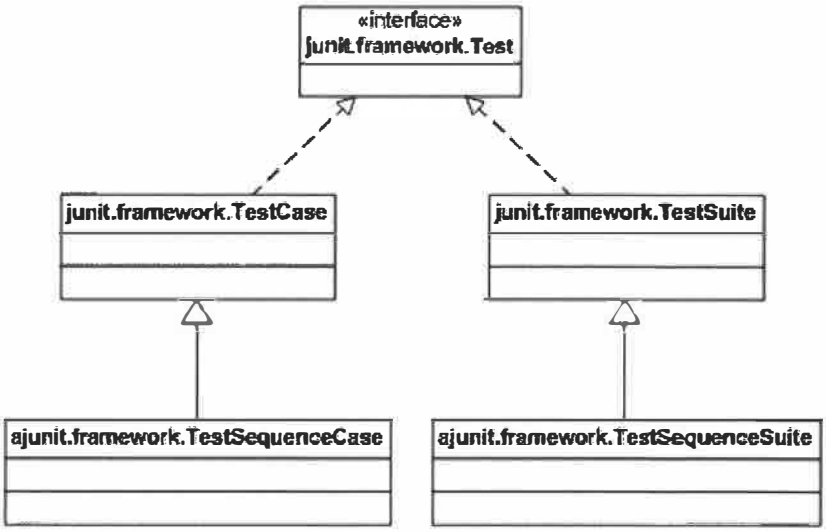

Figure 21 Diagramme UML de TestSequenceCase et TestSequenceSuite

CHAPITRE 8

ÉTUDES DE CAS

Nous présentons, dans ce chapitre, deux études de cas. La première présente le cas d'un aspect unique et la seconde démontre le fonctionnement itératif de notre méthode et de l'outil. Voici une capture d'écran de l'outil qui permet de saisir ces informations.

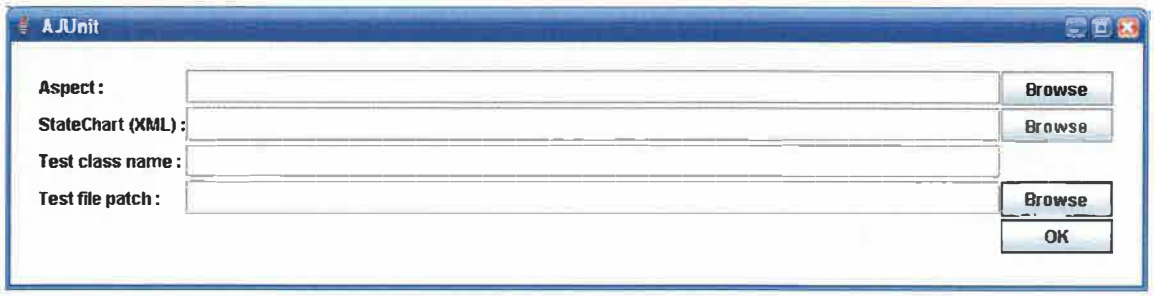

Figure 22 Outil

À partir de cet écran, on sélectionne le fichier contenant l'aspect, le fichier xml contenant le diagramme d'états, le nom de la classe de test qui sera générée ainsi que le répertoire où créer le fichier. Cet écran sera éventuellement appelé à devenir un plug-in pour l'IDE Eclypse.

8.1 Exemple conceptuel

Code de la classe :

```
public class MyClass {
        public int A () {...}

        public void B () {...}

        public void C () {...}

        public void D () {...}

        public int E () {...}

        public void F () {...}

        public void G () {...}

        public void H () {...}

        public void I () {...}

}
```

Tableau 3 Code source de l'exemple conceptuel

Cette classe simple contient 9 méthodes publiques. Toutes les méthodes ont un type de retour void sauf deux : A et E qui retournent une valeur de type entier.

Code de l'aspect :

```
public aspect ExampleAspect {

        pointcut OnAandE() : call(public int *()) && target(MyClass);

        after() : OnAandE () {
                System.out.println("After OnAandE");
        }

}
```

Tableau 4 Code source de l'aspect de l'exemple conceptuel

L'aspect, quant à lui, contient un pointcut OnAandE qui vise les méthodes publiques ayant comme valeur de retour un entier et qui appartiennent à la classe MyClass. Ainsi, dans cet exemple seules les méthodes A et E sont visées par ce pointcut. De plus, l'aspect contient un advice de type after pour les jointpoints définis par le pointcut OnAandE. Cet advice ne fait que laisser une trace d'exécution dans la console.

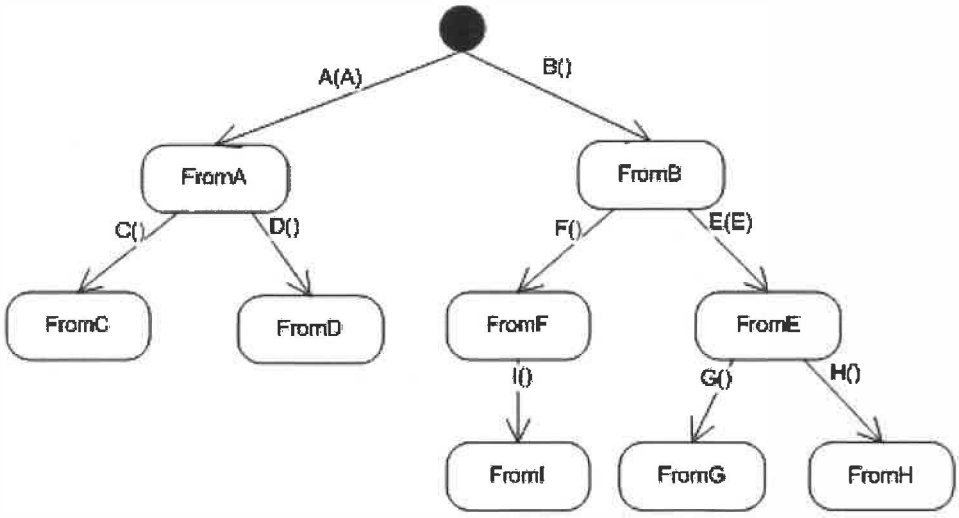

Figure 23 Arbre d'appels des méthodes

Le diagramme d'états de la classe est, en fait, un arbre. Il n'y a pas de chemin circulaire. Cette structure permet de réaliser un exemple simple qui couvre plusieurs cas. Dans cet exemple, ce sont les méthodes A et E qui sont touchées. Ainsi, selon le critère de transition, les transitions de l'état de départ vers l'état FromA et de l'état FromB vers

FromE devraient être testées. De plus, selon le critère de séquence toutes les séquences contenant une transition touchée devraient être testées. Dans ce cas, il s'agit des séquences contenant les transitions A et E, donc les séquences :

- A, C
- A, D
- B, E, G
- B, E, H

Le troisième critère vise les différentes exécutions possibles des advices. Puisque dans ce cas le seul advice ne comprend qu'une seule exécution possible, ce critère ne s'applique pas dans ce cas. Le quatrième critère ne s'applique pas non plus, car il vise les situations où plusieurs aspects affectent une classe.

Résultat de l'analyse

```
public class ExempleTestSequence extends TestSequenceCase {

	public static void main(String[] args) {
		ajunit.sequenceTextui.TestRunner.run(ExempleTestSequence.class);
	}

	public void testD() {
	  assertTrue(true);
	}
	public void testH() {
	}
	public void testF() {
	}
	public void testE() {
	}
	public void testG() {
	}
	public void testC() {
	}
	public void testA() {
		assertEquals("message", 1, 0);
	}
	public void testB() {
	}
	public void testI() {
	}
	public void sequence1() {
		testA();
		testC();
	}
	public void sequence2() {
		testA();
		testD();
	}
	public void sequence3() {
		testB();
		testE();
		testG();
	}
	public void sequence4() {
		testB();
		testE();
		testH();
	}
}
```

Tableau 5 Résultat de l'analyse

Nous pouvons voir dans le résultat que quatre séquences ont été créées et qu'il s'agit des quatre séquences que nous avions identifiées comme devant être testées selon les deux premiers critères de test.

Les étapes précédentes sont propres à AJUnit : l'analyse de l'aspect, relever les interactions avec la classe, produire un arbre afin de faire ressortir l'ensemble des séquences d'exécution de la classe touchée par l'aspect. Ensuite, la création du stub de la classe de test de AJUnit diffère légèrement d'un outil qui génère un stub de classe de test JUnit simplement car AJUnit y ajoute les tests de séquences. La suite est identique à la manière de travailler de JUnit. Il revient au testeur d'y inscrire les tests qui couvriront les cas d'exécution attendus selon les spécifications du système. La ligne de commande dans la méthode testA a été ajoutée manuellement et ne fait pas partie du résultat de l'analyse.

Résultats

On peut remarquer que la méthode testA contient une vérification qui échouera, puisqu'une instruction s'assure que 1 est égal à 0. Puisque les séquences 1 et 2 contiennent un appel à la méthode test, ces deux séquences échoueront en toutes circonstances. Si on exécute cette classe, nous obtiendrons ceci :

```
There were 2 failures:
1)  sequence1(example.ExempleTestSequence)junit.framework.AssertionFailedError:  message
expected:<1> but was:<0>
        at example.ExempleTestSequence.testA(ExempleTestSequence.java:30)
        at example.ExempleTestSequence.sequence1(ExempleTestSequence.java:39)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at ajunit.sequenceTextui.TestRunner.run(TestRunner.java:48)
        at example.ExempleTestSequence.main(ExempleTestSequence.java:8)
2)  sequence2(example.ExempleTestSequence)junit.framework.AssertionFailedError:  message
expected:<1> but was:<0>
        at example.ExempleTestSequence.testA(ExempleTestSequence.java:30)
```

```
        at example.ExempleTestSequence.sequence2(ExempleTestSequence.java:43)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at ajunit.sequenceTextui.TestRunner.run(TestRunner.java:48)
        at example.ExempleTestSequence.main(ExempleTestSequence.java:8)

FAILURES!!!
Tests run: 4,  Failures: 2,  Errors: 0
```

Alors, que si on change l'instruction en cause par assertEquals("message", 1, 1); le résultat sera :

```
Time: 0

OK (4 tests)
```

Le résultat est très semblable à un test JUnit. Il s'agit, en fait, d'une extension du TextRunner de JUnit pour prendre en compte les extensions ajoutées dans AJUnit. Dans la méthode main de la classe de test de la figure 10 on retrouve la ligne :

ajunit.sequenceTextui.TestRunner.run(ExempleTestSequence.class);

C'est cette ligne qui permet d'exécuter l'ensemble des séquences de test. Le TestRunner prend en paramètre un objet respectant l'interface TestCase. N'importe quel Runner de test JUnit peut très facilement être étendu pour obtenir un Runner de test AJUnit compatible avec l'environnement du Runner d'origine, par exemple un plug-in pour l'IDE Eclipse.

Pour avoir une vision du processus, nous pouvons nous référer à cet exemple. Nous avons dans le tableau 6 le code de l'aspect StackAspect qui affecte la classe Stack du framework de Java. Cet aspect prévient qu'un object ne soit présent deux fois dans une même pile. Cependant, le code contient une erreur à la ligne 6, au lieu de permettre l'ajout de l'objet lorsqu'il n'est pas présent, il ne le permet que lorsqu'il l'est déjà, une

simple erreur entre les touches '<' et '>'.

```
1  public aspect StackAspect
2  {
3     Object around ( Object obj, java.util.Stack stack )
4        call(public Object java.util.Stack.push(Object)) && args(obj) && target(stack)
5     {
6        if ( stack.search(obj) > 0 ) //Error here
7           return proceed(obj, stack);
8        else
9           return obj;
10    }
11 }
```

Tableau 6 Code de l'aspect StackAspect

L'outil permet de générer la classe de test présentée dans le tableau 7. L'utilisateur y a inscrit un test pour la méthode « push » qui est utilisé par les deux séquences où est appelée la méthode.

```
public class StackAJUnitTestSequence extends TestSequenceCase  {
        public static void main(String[] args) {
                ajunit.sequenceTextui.TestRunner.run(StackAJUnitTestSequence.class);
        }

        public void testpush() {
                Stack stack = new Stack();
                Integer five = new Integer(5);
                assertTrue(stack.isEmpty());
                stack.push(five);
                stack.push(five);
                assertEquals(1,stack.size());
        }
        public void testpop() {
        }
        public void sequence1() {
                testpush();
                testpop();
        }
        public void sequence2() {
                testpush();
                testpush();
                testpop();
                testpop();
        }
}
```

Tableau 7 Code de la classe de test généré avec un test pour la méthode push

L'exécution du test donne le résultat suivant :

```
There were 2 failures:
1) sequence1(test.StackAJUnitTestSequence)junit.framework.AssertionFailedError:
expected:<1> but was:<0>
        at test.StackAJUnitTestSequence.testpush(StackAJUnitTestSequence.java:20)
        at test.StackAJUnitTestSequence.sequence1(StackAJUnitTestSequence.java:26)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at ajunit.sequenceTextui.TestRunner.run(TestRunner.java:48)
        at test.StackAJUnitTestSequence.main(StackAJUnitTestSequence.java:10)
2) sequence2(test.StackAJUnitTestSequence)junit.framework.AssertionFailedError:
expected:<1> but was:<0>
        at test.StackAJUnitTestSequence.testpush(StackAJUnitTestSequence.java:20)
        at test.StackAJUnitTestSequence.sequence2(StackAJUnitTestSequence.java:30)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at ajunit.sequenceTextui.TestRunner.run(TestRunner.java:48)
        at test.StackAJUnitTestSequence.main(StackAJUnitTestSequence.java:10)

FAILURES!!!
Tests run: 2,  Failures: 2,  Errors: 0
```

On voit, par la description des vérifications, que la pile reste vide alors qu'elle devrait contenir 1 élément dans chacun des cas. Si l'on corrige le code, les tests s'exécuteront avec succès et nous aurons alors le résultat :

```
OK (2 tests)
```

8.2 Étude de cas simple

Cette première étude de cas présente un cas concret d'une classe qui offre un service de recherche aux utilisateurs authentifiés. Il contient les méthodes : Connect, Login, GotoSearch, Search et Logout.

On peut résumer l'exemple par un système de recherche d'une base de connaissances. L'utilisateur doit entrer son nom d'utilisateur ainsi que son mot de passe pour pouvoir effectuer des recherches. À toutes les étapes, l'usager peut mettre fin à sa session en se déconnectant.

À cela il s'ajoute un aspect qui permettra de tenir un historique des tentatives de connexion infructueuses, d'enregistrer dans un journal lorsqu'un usager se déconnecte ainsi que de lister les demandes de recherche non autorisées (dans notre cas lorsque le résultat de la recherche est null).

Voici le diagramme d'états de la classe ainsi que son code :

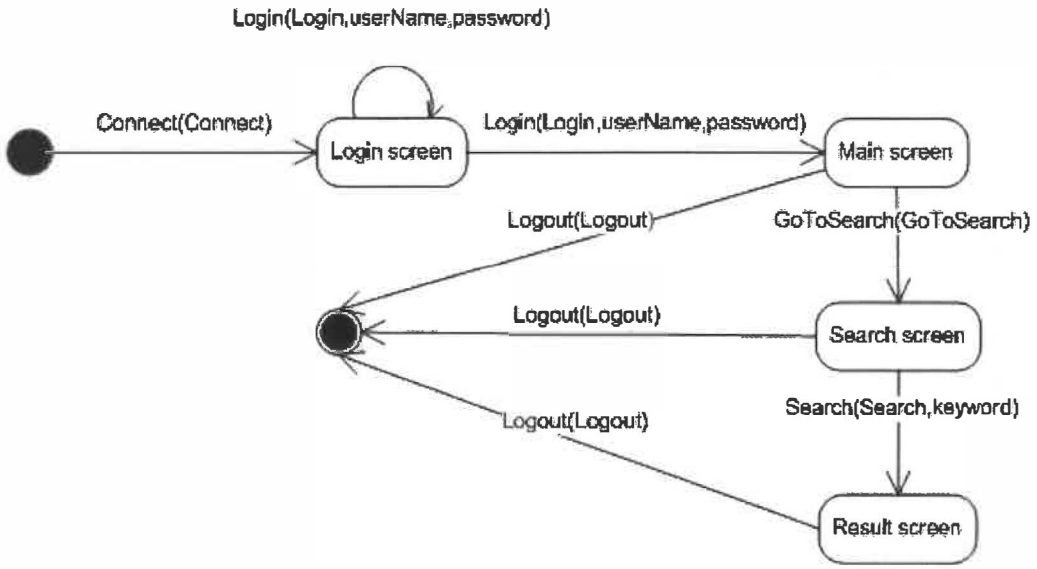

Figure 24 Diagramme d'états de l'exemple à un aspect

Ainsi, lorsque l'usagé entre dans le système, il arrive d'abord à l'écran de connexion où il devra fournir son nom d'usager ainsi que son mot de passe pour poursuivre. S'il ne fournit pas des informations valides, il devra entrer de nouveau ces informations. Une fois identifié, l'usager se retrouve sur la page d'accueil d'où il pourra aller faire une recherche dont le résultat lui sera présenté sur l'écran de résultat des recherches. De plus, une fois connecté, l'usager peut se déconnecter à n'importe quelle étape.

La classe d'exemple représente un système de recherche où l'usager doit s'identifier afin d'avoir accès au service de recherche. Le code est ci-dessous :

```
public class Example2 {
   public static int Disconnected = 0;
   public static int Connected    = 1;
   public static int Logging      = 2;
   public static int Logged       = 3;
   public static int Searching    = 4;
   private int m_state;

   public Example2()    { m_state = Disconnected; }
   public int GetState()   { return m_state; }
   public void Connect()  { m_state = Connected; }

   public boolean Login(String username, String password) {
      m_state = Logging;
      if(username.equals(password)) {
         m_state = Logged;
      }
      else {
         m_state = Connected;
      }
      return m_state == Logged;
   }

   public void Logout() { m_state = Disconnected; }

   public String Research(String keyword) {
      if(m_state == Logged) {
         return Search(keyword);
      }
      else {
         return null;
      }
   }

   public void GoToSearch() { /*Navigate to the page...*/ }

   private String Search(String keyword) {
      m_state = Searching;
      //Sreaching...
      m_state = Logged;
      return "result";
   }
}
```

Tableau 8 Code source de la classe de l'exemple à un aspect

Le code de l'exemple a été écrit de façon à pouvoir être testé facilement. Par exemple, la méthode Search retourne toujours « result » comme résultat et la méthode Login considère le couple username/password valide si le deux sont égaux. C'est sur ce genre de comportement que nos tests seront basés pour déterminer le comportement attendu de la classe.

L'aspect possède trois advices qui inscrivent tous un élément dans le journal d'évènements. Le premier de type before est lié au pointcut OnLogout et permet d'inscrire dans le journal la déconnexion. Le second, quant à lui, est de type after et est lié au pointcut OnLogin, mais cet advice ne laisse une trace d'exécution que si l'authentification de l'usager a échoué. Le dernier aspect aussi de type after permet de tracer les demandes de recherche non autorisées. Voici le code de l'aspect :

```
public aspect Example2Aspect {

  pointcut OnLogout() : call(public void Logout()) && target(Example2);

  pointcut OnLogin(String username, String password) : call(public boolean Login(String, String))
            && target(Example2)
            && args(username, password);

  pointcut OnResearch(String keyword) : call(public String Research(String))
            && target(Example2)
            && args(keyword);

  before() : OnLogout () {
    System.out.println("Before Logout");
  }

  after(String username, String password) returning (boolean result) : OnLogin(username, password) {
    if (!result)
      System.out.println("Login fail : " + username + ", " + password);
  }

  after(String keyword) returning (String researchResult) : OnResearch(keyword) {
    if(researchResult == null)
      System.out.println("Unautorised research on : " + keyword);
  }
}
```

Tableau 9 Code source de l'aspect de l'exemple à un aspect

AJUnit commencera par analyser le code de l'aspect ainsi que l'analyse comparative avec le diagramme d'états de la classe. On peut voir ici l'arbre qui sera généré par l'outil avec les méthodes ciblées par l'aspect en rouge :

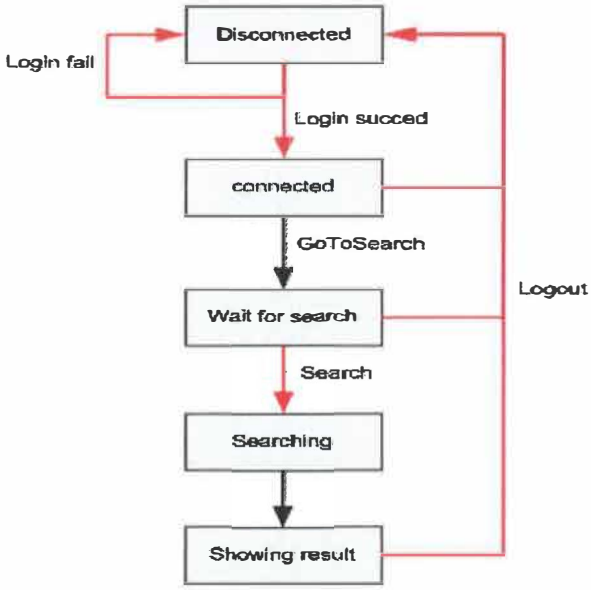

Figure 25 Arbre de séquences

On peut voir, en comparant le code de l'aspect au diagramme d'états, que cinq transitions devront être testées selon le premier critère. Trois de ces transitions sont les trois transitions Logout pour le premier advice et les deux autres sont les deux chemins possibles pour la méthode Login couverts par le second advice. Si on se réfère au deuxième critère de test, six séquences devront être couvertes. Il s'agit de deux séquences pour chacune des trois transitions Logout, une si l'authentification est réussie du premier coup et une autre si ce n'est pas le cas. On peut voir ces séquences listées ici et dans la classe de test générée :

- Connect, Login(Success), Logout
- Connect, Login(Failure), Login(Success), Logout
- Connect, Login(Success), GoToSearch, Logout
- Connect, Login(Failure), Login(Success), GoToSearch, Logout
- Connect, Login(Success), GoToSearch, Search, Logout
- Connect, Login(Failure), Login(Success), GoToSearch, Search, Logout

Cette analyse permettra de générer le diagramme d'états étendu duquel les séquences devant être testées seront extraites. De ces séquences un stub sera généré par l'outil. Il comprend les six séquences à tester ainsi qu'un stub pour chaque méthode qui fait partie d'au moins une séquence :

```
public class Exemple2TestSequence extends TestSequenceCase {
public static void main(String[] args) { ajunit.sequenceTextui.TestRunner.run(Exemple2TestSequence.class); }
public void testLogin() { }
public void testResearch() { }
public void testLogout() { }
public void testConnect() { }
public void testGoToSearch() { }
public void sequence1() {
        testConnect();
        testLogin();
        testLogin();
        testGoToSearch();
        testResearch();
        testLogout();
}
public void sequence2() {
        testConnect();
        testLogin();
        testLogin();
        testGoToSearch();
        testLogout();
}
public void sequence3() {
        testConnect();
        testLogin();
        testLogin();
        testLogout();
}
public void sequence4() {
        testConnect();
        testLogin();
        testGoToSearch();
        testResearch();
        testLogout();
}
public void sequence5() {
        testConnect();
        testLogin();
        testGoToSearch();
        testLogout();
}
public void sequence6() {
        testConnect();
        testLogin();
        testLogout();
}
}
```

Tableau 10 Code source de la classe de test résultante

Une fois que l'on inscrit le code des tests, le stub pour la méthode Login est dupliqué pour obtenir deux versions, une où l'indentification réussie et une autre où elle échoue, pour simuler les deux cas de figure qui se retrouvent dans les séquences selon le diagramme d'états. Voici ce que l'on obtient.

```
public class Exemple2TestSequence extends TestSequenceCase {

public static void main(String[] args) {
        ajunit.sequenceTextui.TestRunner.run(Exemple2TestSequence.class);
}

private Exemple2 m_class;

public void setUp()
{
   m_class = new Exemple2();
}

public void tearDown()
{
   m_class = null;
}

public void testConnect() {
   assertEquals(Exemple2.Disconnected ,m_class.GetState());
   m_class.Connect();
   assertEquals(Exemple2.Connected ,m_class.GetState());
}

public void testResearch() {
   assertEquals(Exemple2.Logged, m_class.GetState());
   assertEquals("result", m_class.Research("keyword"));
   assertEquals(Exemple2.Logged ,m_class.GetState());
}

public void testLoginSucces() {
   assertEquals(Exemple2.Connected ,m_class.GetState());
   m_class.Login("succes", "succes");
   assertEquals(Exemple2.Logged, m_class.GetState());
}

public void testLoginFail() {
   assertEquals(Exemple2.Connected ,m_class.GetState());
   m_class.Login("login will","fail");
   assertEquals(Exemple2.Connected, m_class.GetState());
}
```

```
public void testGoToSearch() {
   assertEquals(Exemple2.Logged ,m_class.GetState());
   m_class.GoToSearch();
   assertEquals(Exemple2.Logged ,m_class.GetState());
}

public void testLogout() {
   m_class.Logout();
   assertEquals(Exemple2.Disconnected, m_class.GetState());
}

public void sequence1() {
   System.out.println("sequence 1");
            testConnect();
            testLoginFail();
            testLoginSucces();
            testGoToSearch();
            testResearch();
            testLogout();
}

public void sequence2() {
   System.out.println("sequence 2");
            testConnect();
            testLoginFail();
            testLoginSucces();
            testGoToSearch();
            testLogout();
}

public void sequence3() {
   System.out.println("sequence 3");
            testConnect();
            testLoginFail();
            testLoginSucces();
            testLogout();
}

public void sequence4() {
   System.out.println("sequence 4");
            testConnect();
            testLoginSucces();
            testGoToSearch();
            testResearch();
            testLogout();
}

public void sequence5() {
   System.out.println("sequence 5");
```

```
            testConnect();
            testLoginSucces();
            testGoToSearch();
            testLogout();
}

public void sequence6() {
   System.out.println("sequence 6");
            testConnect();
            testLoginSucces();
            testLogout();
}
}
```

Tableau 11 Classe de test avec les tests de l'usagé

Lors de l'exécution nous avons :

```
.sequence 1
Login fail : login will, fail
Before Logout
.sequence 2
Login fail : login will, fail
Before Logout
.sequence 3
Login fail : login will, fail
Before Logout
.sequence 4
Before Logout
.sequence 5
Before Logout
.sequence 6
Before Logout

Time: 0,031

OK (6 tests)
```

On peut voir que l'outil a ressorti les six séquences d'exécution que les critères décrivent comme devant être testées.

Continuité avec JUnit

Puisque notre outil hérite de JUnit, les classes de test AJUnit sont également des classes de tests JUnit comme nous l'avons vu dans la figure 7. Donc, si on exécute la classe de test que nous avons générée comme elle était dans la figure 12 comme un test JUnit, nous aurons comme résultat dans le plug-in JUnit de Eclipse :

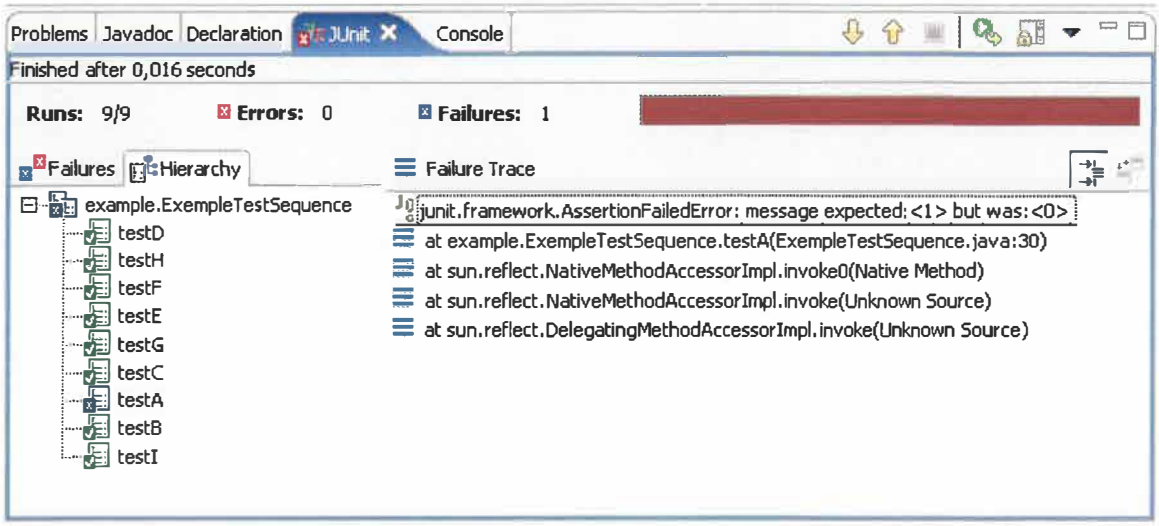

Figure 26 Capture d'écran du résultat d'un test JUnit sur la classe de test AJUnit

Cependant, JUnit ne considère comme test unitaire que les méthodes dont le nom commence par le préfixe « test ». Donc, les méthodes de séquence de test commençant par le préfixe « sequence » ne seront pas considérées.

8.3 Étude de cas à plusieurs aspects

La seconde étude de cas présente le fonctionnement de notre méthode lorsque plusieurs aspects affectent une même classe. Puisque le test des couples AspectA-ClassA et AspectB-ClassA n'est pas l'équivalent du test du trio AspectA-AspectB-ClassA, réaliser deux tests à un aspect n'est pas suffisant. C'est pourquoi notre méthode est incrémentale, c'est-à-dire que nous pouvons inclure un aspect supplémentaire sur un test déjà généré pour tester à la fois ce nouvel aspect, son intégration à la classe mais également aux autres aspects déjà inclus.

Pour cette étude de cas, nous présentons une classe dont le diagramme d'états est volontairement simple mais qui possède une particularité très importante. Chacune des deux méthodes qui seront affectées par un aspect différent ne se retrouve jamais dans la même séquence d'exécution. Cette particularité est nécessaire pour éviter que les séquences de test pour un aspect n'englobent également les séquences de test du second aspect et que l'intégration du second aspect ne puisse être observée puisqu'il n'introduit aucune nouvelle séquence de test.

La classe utilisée pour l'étude de cas est une classe de parsing dont la source de donnée peut être un fichier local ou accessible par réseau. Voici son diagramme d'états :

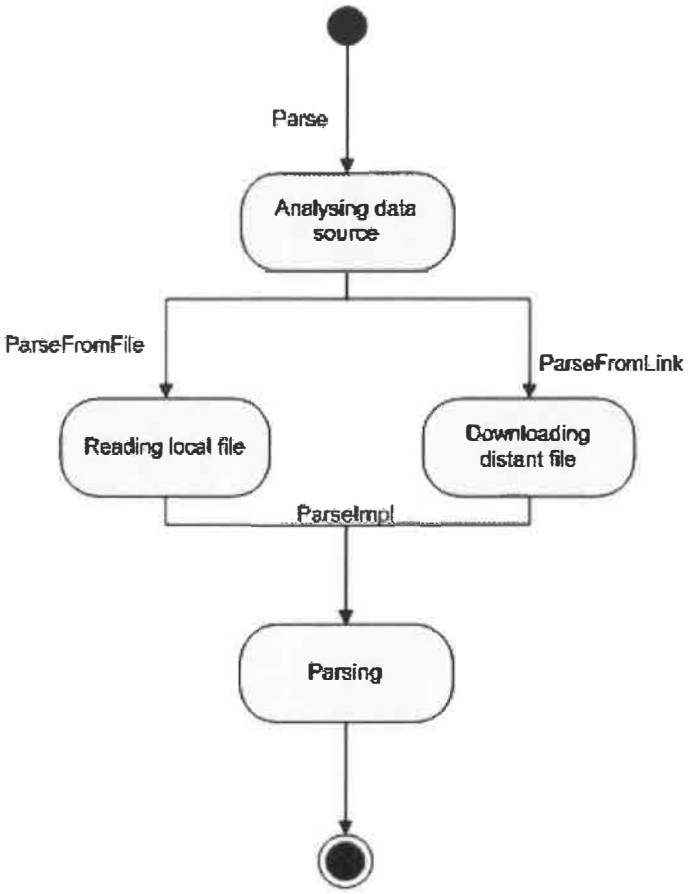

Figure 27 Diagramme d'états de l'exemple à plusieurs aspects

La seconde étude de cas reprend la première mais y ajoute un second aspect. Ce second aspect s'occupe de garder en cache les résultats de recherche pour accélérer le traitement de recherche qui ont déjà été effectués. Le code de l'aspect est présenté dans le tableau 12.

Le premier aspect que nous allons introduire permet de s'assurer qu'un fichier local qui est couvert par un service de contrôle de source est bien à jour avec la version du serveur avant de faire quoique ce soit avec ce fichier pour ainsi éviter de travailler avec des versions obsolètes. Voici le code de cet aspect :

```
public aspect SourceControlAspect {

   pointcut FileOpen(String filename) : call(public void ParseFromFile(String))
   && target(Example3)
   && args(filename);

   before(String filename) : FileOpen(filename) {
      //Get last version
   }
}
```

Tableau 12 Code source de l'aspect de contrôle de source

On peut voir que cet aspect touche la méthode ParseFromFile de notre classe exemple. Si on génère la classe de test avec les séquences de test qui doivent être testées nous obtenons :

```
public class Example3WithSourceControlTestSequence extends TestSequenceCase  {

   public static void main(String[] args) {
      ajunit.sequenceTextui.TestRunner.run(Example3WithSourceControlTestSequence.class);
   }

   public void testParse() {

   }
   public void testParseImpl() {

   }
   public void testParseFromLink() {

   }
   public void testParseFromFile() {

   }
   public void sequence1() {
      testParse();
      testParseFromFile();
      testParseImpl();
   }
}
```

Tableau 13 Code source de la classe de test pour l'aspect de contrôle de source

Ce résultat est généré dans le processus par le générateur de séquences que l'on peut voir au point 3 dans le figure 28. Mais, une des informations intermédiaires du processus

devient très importante pour ajouter des aspects supplémentaires à notre classe de test. Il s'agit du diagramme d'états étendu généré par l'analyseur d'impact au point 2.

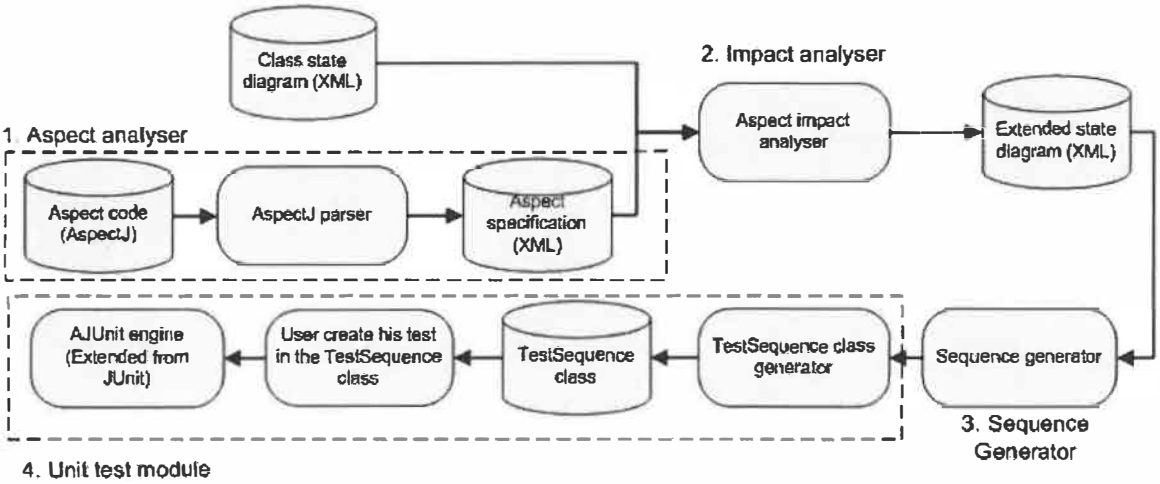

Figure 28 Diagramme du module d'analyse

C'est de ce diagramme d'états étendu qui comprend l'information relative à l'intégration du premier aspect de contrôle de source à la classe que nous allons repartir pour l'intégration du second aspect.

La figure 3 reproduite ci-dessous illustre ce procédé. Le diagramme d'états étendu produit par l'analyseur d'impact des aspects peut être réutilisé en entrée. Dans un tel cas, l'information de l'influence de l'aspect analysé présentement sera simplement ajoutée à celle des aspects analysés précédemment et un nouveau diagramme sera généré.

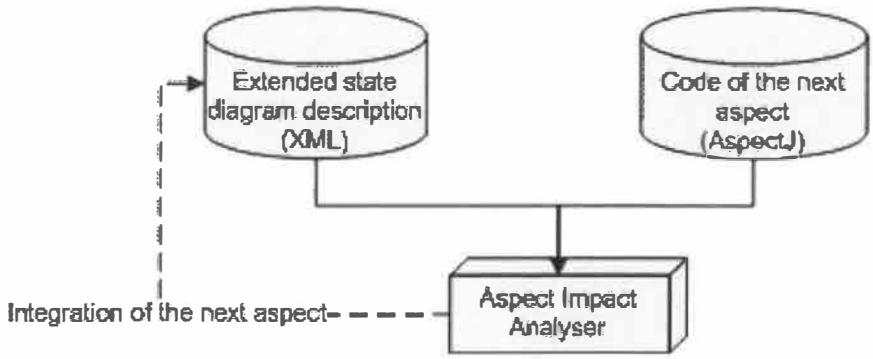

Figure 29 Intégration incrémentale des aspects

Le second aspect s'intègre à une méthode de la classe qui ne fait partie d'aucune des séquences de test couvertes par le test du premier aspect. Cet aspect a pour but de s'assurer de la sécurité des fichiers téléchargés depuis le réseau et s'intéresse donc à la méthode ParseFromLink, voici son code :

```
public aspect VirusCheckAspect {

   pointcut Download(String filename) : call(public void ParseFromLink(String))
   && target(Example3)
   && args(filename);

   void around(String link) : Download(link) {
      //Donwload file, if file secure proceed jointpoint
   }
}
```

Tableau 14 Code source de l'aspect de vérification de la sécurité des fichiers

En générant, la classe de test avec cet aspect, en partant du diagramme d'états étendu généré lors de l'intégration du premier aspect, nous obtenons cette classe de test :

```
import ajunit.framework.TestSequenceCase;

public class Example3WithBothTestSequence extends TestSequenceCase  {

   public static void main(String[] args) {
      ajunit.sequenceTextui.TestRunner.run(Example3WithBothTestSequence.class);
   }

   public void testParseImpl() {

   }
   public void testParse() {

   }
   public void testParseFromFile() {

   }
   public void testParseFromLink() {

   }
   public void sequence1() {
      testParse();
      testParseFromFile();
      testParseImpl();
   }
   public void sequence2() {
      testParse();
      testParseFromLink();
      testParseImpl();
   }
}
```

Tableau 15 Code de la classe de test pour les deux aspects

On peut voir que la séquence de test Parse > ParseFromLink > ParseImpl est couverte puisque le second aspect affecte la méthode ParseFromLink, mais en plus la séquence Parse > ParseFromFile > ParseImpl est également couverte bien que cette séquence ne soit pas affectée par l'aspect. Cette séquence est présente car l'information quant à l'intégration du premier aspect est toujours présente.

C'est ainsi que notre méthode permet de réaliser des tests qui incluent l'influence de l'ensemble des aspects pour ainsi détecter non seulement les problèmes entre la classe et

l'aspect mais également entre les aspects dans les différentes classes. Mais, la procédure incrémentale permet de commencer par des tests avec une portée plus restreinte et d'augmenter la complexité de façon contrôlée.

CONCLUSION

La technologie aspect offre de nouveaux outils pour mieux représenter et découper les différentes composantes d'un programme. Cependant, de nombreux outils développés pour les systèmes orientés objet ne permettent pas de couvrir les nouvelles fonctionnalités du paradigme aspect. C'est le cas des techniques de test unitaire des classes qui ne couvrent pas le test des aspects seuls ou entrelacés dans des objets.

Notre technique introduit une série de critères de test permettant de définir quels sont les éléments qui doivent être couverts pas des tests lorsqu'un aspect affecte une classe. À partir de ces critères, nous avons développé un outil de génération de tests unitaires de classes affectées par un ou plusieurs aspects. En ce basant sur le comportement dynamique de la classe, extrait depuis sont diagramme d'états, et du code source des aspects, notre outil permet de générer des classes de test unitaire couvrant les critères de test que nous avons défini.

Une extension du framework de test unitaire JUnit a été développée pour offrir un support à l'écriture des tests unitaires et à leur exécution. Ce système est composé de classes qui dérivent de leur équivalent JUnit. Cette implémentation permet que les classes de test AJUnit soient également des classes de test JUnit et qu'un grand nombre d'utilitaires compatibles JUnit le soient également avec AJUnit.

Les études de cas que nous avons effectuées présentent le mécanisme de génération de séquences de tests depuis un exemple conceptuel démontrant comment l'arbre des appels de méthode est extrait du diagramme d'états de classe, pour être ensuite comparé à la définition d'un aspect pour déterminer quelles sont les séquences qui doivent figurer dans la classe de test générée. Les études de cas qui ont suivi présentent un cas réel d'une classe affectée par un aspect ainsi qu'une classe affectée par plusieurs aspects. La dernière étude de cas explique la technique de génération de tests incrémentale lorsque plusieurs aspects affectent une même classe.

BIBLIOGRAPHIE

[Alexander01] Roger T. Alexander, James M. Bieman (2002). Challenges of Aspect-oriented Technology. ICSE 2002 Workshop on Software Quality.

[Alexander02] 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, Colorado, USA. Technical Report CS-4-105.

[Anbalagan01] 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.

[AspJ] AspectJ Projet, [En ligne] http://eclipse.org/aspectj/ (Consulté Août 2007).

[Badri01] Mourad Badri, Linda Badri & Maxime Bourque-Fortin (2005). Generating Unit Test Sequences For Aspect-Oriented Programs: Towards A Formal Approach Using UML State Diagrams, 3rd International Conference on Information & Communications Technology (ICICT 2005), Cairo, Egypt.

[Binder01] R. V. Binder (2000). Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley.

[Briand01] L. C. Briand, Y. Labiche, Y. Wang (2004). Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts. Proc. of ACM International Conference on Software Engineering (ICSE'04), pp. 86-95, Edinburgh, Scotland, UK.

[Elrad01] Elrad, T., R.E. Filman, A. Bader (2001). Aspect-oriented programming: Introduction. Communications of the ACM. 44(10): p. 29-32.

[Mahoney01] Mark Mahoney, Atef Bader, Tzilla Elrad, Omar Aldawud (2004). Using Aspects to Abstract and Modularize Statecharts. The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004.

[Kandé01] Mohamed M. Kandé, Jörg Kienzle, Alfred Strohmeler (2000). From AOP to UML : Toward an aspect-oriented architectural modeling approach. UML'2000 - The Unified Modeling Language: Advancing the Standard, hird International Conference, York, UK.

[Lemos01] Otávio Augusto Lazzarini Lemos, José Carlos Maldonado, Paulo Cesar Masiero (2004). Data Flow Integration Testing Criteria for Aspect-Oriented Programs. Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos.

[Offutt01] Jeff Offutt and, Aynur Abdurazik (1999). Generating Tests from UML Specifications. Second International Conference on the Unified Modeling Language (UML99), pages 416-429, Fort Collins, CO.

[Offutt02] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, Paul Ammann (2003). Generating test data from state-based specifications. The Journal of Software Testing, Verification and Reliability, 13(1):25-53.

[Offutt03] Jeff Offutt, Yiwei Xiong and Shaoying Liu (1999). Criteria for Generating Specification-based Tests. Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99), pages 119-131.

[Pawlak01] Renaud Pawlak, Housam Younessi (2004). On getting use cases and aspects to work together. Journal of Object Technology Vol. 3, No. 1, January-February 2004.

[Sullivan01] Kevin Sullivan, Lin Gu, Yuanfang Cai (2002). Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ. Proceedings of the 1st international conference on Aspect-oriented software development. Enschede, The Netherlands.

[Suzuki01] Junichi Suzuki, Yoshikazu Yamamoto (1999) Extending UML with Aspects : Aspect support in the design phase. ECOOP 1999.

[Videira01] Cristina Videira Lopes and Trung Chi Ngo. UnitTesting Aspectual Behavior.

[Vieira01] Marlon E. Vieira, Marcio S. Dias, Debra J. Richardson (2000). Object-Oriented Specification-Based Testing Using UML Statechart Diagrams. Proceedings of the Workshop on Automated

Program Analysis, Testing and Verification at ICSE 2000.

[Xu01] Dianxiang Xu, Weifeng Xu, Kendall Nygard (2005). A State-Based Approach to Testing Aspect-Oriented Programs. In Proc. of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), July 14-16, Taiwan

[Xu02] 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.

[Zakaria01] Aida Atef Zakaria, Dr. Hoda Hosny, Dr. Amir Zeid (2002). A UML Extension for Modeling Aspect-Oriented Systems. Second International Workshop on Aspect-Oriented Modeling with UML.

[Zhao01] 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.

[Zhao02] Jianjun Zhao (2002). Tool Support for Unit Testing of Aspect-Oriented Software. OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development, Seattle, WA, USA.

[Zhao03] Jianjun Zhao : *Unit Testing for Aspect-Oriented Programs.* Informatik - Forschung und Entwicklung. Volume 20, Numbers 1-2 p.57-71 October, 2005

[Zhao04] Jianjun Zhao, Tao Xie, Nan Li (2006). Towards Regression Test Selection for AspectJ Programs. International Symposium on Software Testing and Analysis Portland, Maine, USA.