UNIVERSITÉ DU QUÉBEC

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

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
PHILIPPE MASSICOTTE

TEST ORIENTÉ ASPECT : UNE APPROCHE FORMELLE BASÉE SUR LES DIAGRAMMES DE COLLABORATION

TROIS-RIVIÈRES
Janvier 2006

TEST ORIENTÉ ASPECT : UNE APPROCHE FORMELLE BASÉE SUR LES DIAGRAMMES DE COLLABORATION

Philippe Massicotte

SOMMAIRE

Le développement orienté aspect est un émergeant paradigme de programmation. Il permet, essentiellement, d'améliorer la séparation des préoccupations dans un programme. Il introduit, en fait, de nouvelles abstractions dans le développement de logiciel. AspectJ, comme langage de programmation orientée aspect, est une extension orientée aspect du langage Java. Les langages de programmation orientée objet présentent, en effet, de sérieuses limitations quant à la représentation des préoccupations dans un programme. Plusieurs préoccupations transversales se retrouvent ainsi dispersées dans plusieurs classes dans un système orienté objet. Le paradigme aspect permet de les représenter de façon modulaire dans des unités séparées appelées aspects. Cependant, il introduit de nouvelles dimensions en termes de contrôle et de complexité. Les nouvelles dépendances entre les aspects et les classes apportent de nouveaux challenges en termes de test. En fait, les aspects peuvent interagir avec n'importe quelles classes dans un programme. Les interactions entre les aspects et les classes constituent de nouvelles sources potentielles de fautes. Les techniques de test orienté objet classiques ne couvrent pas les nouvelles dimensions introduites par les aspects. Ainsi, de nouvelles techniques de test doivent être développées. Nous présentons, dans ce mémoire, une nouvelle technique de test orienté aspect. Elle permet la génération de séquences de test basées sur les interactions dynamiques entre les aspects et les classes (analyse statique), d'une part, et la vérification de leur exécution (analyse dynamique), d'autre part. Nous nous concentrons, en particulier, sur l'intégration d'un ou de plusieurs aspects au sein d'une collaboration entre un groupe d'objets. Nous introduisons également une série de critères de test appropriés. L'approche proposée suit un processus itératif. Nous avons également développé un outil dans le but de supporter automatiquement notre technique.

ASPECT ORIENTED TESTING: A FORMAL APPROACH BASED ON COLLABORATION DIAGRAMS

Philippe Massicotte

ABSTRACT

Aspect-Oriented Software Development is an emerging software engineering paradigm. It provides new constructs and tools to improve separation of crosscutting concerns into single units called aspects. The aspect paradigm introduces, in fact, new abstractions in software development. AspectJ is an aspect-oriented extension for Java. Actually, existing object-oriented programming languages suffer, in fact, from a serious limitation in modularizing adequately crosscutting concerns. Many concerns crosscut several classes in an object-oriented system. However, the aspect paradigm introduces new dimensions in terms of control and complexity. New dependencies between aspects and classes result in new testing challenges. In fact, aspects can interact with any class in a program. Interactions between aspects and classes are new sources for program faults. Object-Oriented Testing techniques do not cover the new dimensions introduced by aspects. Thus, new aspect-oriented testing techniques must be developed. We present a new technique to generate test sequences based on the dynamic interactions between aspects and classes. The technique supports also the verification process. We focus, in particular, on the integration of one or more aspects in the collaboration of a group of objects. We introduce, also, associated testing criteria. The proposed approach follows an iterative process. We developed a tool to automatically support our technique.

REMERCIEMENTS

J'aimerais tout d'abord remercier mes parents de m'avoir supporté tant financièrement que moralement tout au long de mes études universitaires. Sans eux, il m'aurait été impossible de réaliser ce mémoire.

Je remercie également mes directeurs de recherche Mourad Badri et Linda Badri pour leur soutien tout au long de cet ambitieux projet. Leur grande disponibilité ainsi que leur professionnalisme furent certainement la clé du succès dans la réalisation de ce projet de maîtrise.

Finalement, je souhaite remercier tout particulièrement Claudine, ma conjointe, pour son support moral. Elle m'a fourni la force nécessaire pour persévérer, et ce, même dans les moments les plus difficiles.

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

CHAPITRE 1

INTRODUCTION

La programmation orientée objet éprouve des limites sérieuses quant à la représentation des préoccupations transverses dans un programme. Le code correspondant à ces préoccupations est souvent dupliqué à plusieurs endroits dans un programme. Cet enchevêtrement rend l'application difficile à comprendre, à tester, à réutiliser et à maintenir en particulier dans le cas des applications réparties, où le nombre de sous problèmes à traiter est élevé [Balt01]. Le développement orienté aspect *AOSD* [Aosd05] offre de nouvelles perspectives quant à la séparation des préoccupations transverses (*separation of concerns*) dans un programme. Le paradigme aspect permet d'effectuer une bonne factorisation de ces différentes préoccupations en les regroupant dans des unités modulaires appelées aspects [Ajpg02]. Ceci a pour conséquence une réduction de la dispersion du code correspondant et une amélioration de sa modularité [Ajws05]. Malgré les nombreux avantages que le paradigme aspect procure, il ne reste pas moins qu'il n'est pas encore mature. En particulier, plusieurs problèmes sont posés pour le test des programmes orientés aspect. Les aspects apportent de nouvelles abstractions au génie logiciel. Les approches existantes pour le test de logiciels ne couvrent pas les spécificités du paradigme aspect tel que mentionné par plusieurs auteurs [Alex04, Balt01, Mort04].

Le processus de test représente l'une des étapes les plus importantes dans le cycle de développement des logiciels. Il constitue une tâche essentielle dans l'assurance de leur qualité. Dans certains cas, les coûts reliés au test représentent 50 % du coût total de développement [Beiz90]. Le test orienté aspect apporte de nouveaux défis en termes de recherche. Le développement du paradigme aspect ainsi que sa généralisation son liés, entre autres, au développement de techniques et d'outils supportant le test des programmes orientés aspect. Les aspects apportent de nouvelles dimensions en termes de contrôle. Les méthodes actuelles de test orientées objet, ne sont pas adaptées pour la technologie aspect. Le code des aspects ainsi que les

nouveaux mécanismes permettant son intégration au code objet peuvent provoquer de nouvelles fautes [Alex04]. La relation caractéristique qui existe entre les aspects et les classes diffère de celle présente entre les classes dans un système orienté objet. De nouvelles stratégies de test doivent donc être développées pour les systèmes orientés aspect tenant compte de ce niveau d'abstraction supplémentaire.

1.1 Problématique

Les aspects décrivent explicitement où et comment les préoccupations seront intégrées au code objet en utilisant les points de jointure et les advice. La principale problématique vient de la relation entre les aspects et les classes. Les liens reliant un aspect à une classe ne peuvent être identifiés en analysant les classes [Alex04, Balt01, Mort05]. Une des formes majeures de dépendances entre les aspects et les classes vient du fait que le concept de l'appelant et de l'appelé, connu dans les systèmes orientés objet, prend dans les programmes orientés aspect un nouveau sens [Walk99, Zhou04]. La plupart des stratégies de test orienté objet se basent sur ce type de relation entre les classes [Ball98]. Dans un système orienté objet, l'appelant et l'appelé, à un haut niveau, sont des classes. Dans un tel système, un appelant spécifie les différents appels qu'il effectue ainsi que le contrôle lié à ces appels. Alors que dans un système orienté aspect, l'inverse se produit puisque les règles d'intégration sont définies dans les aspects à l'insu des classes. L'aspect décrit, à l'aide de diverses constructions, comment cet appel sera effectué. Ce niveau supplémentaire d'abstraction ainsi que ses conséquences en termes de contrôle doivent être pris en compte afin de s'assurer que les dépendances entre les aspects et les classes soient testées adéquatement [Zhou04].

1.2 Approche

Nous proposons, dans ce mémoire, une série de critères de test relatifs aux nouvelles dimensions introduites par l'intégration des aspects dans un code objet. Nous nous intéressons, en particulier, à l'intégration de plusieurs aspects à un groupe de classes collaborantes. Nous proposons ensuite une stratégie d'intégration aspects/classes

basée sur les critères définis et tenant compte des interactions dynamiques entre les aspects et les classes. La stratégie présentée est incrémentale et itérative. La démarche adoptée consiste à générer, dans un premier temps, les séquences de test correspondant aux différents scénarios de la collaboration entre objets. Cette collaboration est spécifiée à l'aide des diagrammes de collaboration UML. L'intégration des aspects se fait dans un second temps de façon incrémentale. Les nouvelles séquences générées permettent de vérifier l'impact des aspects sur les scénarios du diagramme de collaboration. Nous nous intéressons aux programmes AspectJ. Notre méthode est, cependant, générale et peu être utilisée pour d'autres implémentations de la technologie aspect.

1.3 Organisation

Ce mémoire se divise en douze chapitres. Le chapitre 2 résume l'état de l'art des travaux portant sur le test orienté aspect. Le chapitre 3 traite de la programmation aspect et du langage AspectJ. Nous y présentons les fondements de base de ce paradigme. Les motivations valorisant le développement de cette technologie seront abordées. Une brève introduction à AspectJ est également proposée afin de familiariser le lecteur avec les concepts de base de ce langage. Le chapitre 4 introduit notre stratégie de manière générale. Nous y présentons la méthodologie de l'approche. Dans le chapitre 5, nous discutons des défis qu'apporte ce paradigme au niveau du test. Pour ce faire, nous basons nos réflexions sur divers travaux dans le domaine. Une brève introduction présente les diagrammes de collaboration d'un point de vue global au chapitre 6. Une série de critères de test sont proposés au chapitre 7 en vue du développement de notre stratégie. Ces critères visent principalement à définir précisément ce qui doit être testé. Le chapitre 8 propose deux stratégies possibles en ce qui concerne l'intégration des aspects dans le processus de test. Nous discutons des avantages et des inconvénients de chacune d'elles afin de motiver l'approche que nous avons adoptée. Le processus de génération des séquences de test est décrit en détail au chapitre 9. Cette démarche est mise en application sur un cas réel d'un programme AspectJ. Le chapitre 10 présente la deuxième étape de notre stratégie, soit la vérification automatique des

séquences générées. Nous présentons un outil que nous avons développé pour supporter notre stratégie au chapitre 11. Finalement, une conclusion générale sera présentée au chapitre 12.

CHAPITRE 2

ÉTAT DE L'ART

Le test orienté aspect constitue une tâche importante dans l'assurance qualité des systèmes orientés aspect. Ces dernières années ont vu l'émergence de quelques travaux intéressants dans ce domaine. Ubayashi et Tamai [Ubay02] ont proposé une méthode permettant de vérifier si un programme orienté-aspect satisfait certaines propriétés attendues.

Alexander et al. [Alex04] ont discuté de différentes fautes pouvant se produire dans un système orienté aspect. Ils ont tenu compte des nouvelles dimensions introduites par l'intégration des aspects dans un code objet. Ils proposent un modèle incluant six catégories de fautes permettant d'identifier les sources potentielles d'erreurs dans un système orienté aspect. Ce modèle constitue, à notre avis, une base intéressante qui pourrait orienter le développement de stratégies de test orienté aspect.

Mortensen et al. [Mort04] présentent plusieurs critères de test tenant compte des aspects en identifiant précisément ce qui doit être testé. Leur approche combine, en fait, deux techniques de test traditionnelles : (1) une approche structurelle (*whitebox coverage*) ainsi que (2) le test par *mutation*. Ils classifient les aspects selon qu'ils modifient ou non l'état d'un système pour sélectionner le type de test à effectuer. La technique de mutation est utilisée pour insérer volontairement des fautes dans le programme et vérifier l'efficacité du test choisi. Cette technique consiste principalement à découvrir les fautes relatives au code introduit par les advice. Les opérateurs de mutation sont appliqués au niveau des liens unissant les advice aux classes.

Une approche similaire, mais plus complète est proposée par Deursen et. al. [Deur05]. Leur stratégie de test applique le modèle de faute proposé par Alexander et al. [Alex04] sur trois niveaux. (1) Le test basé sur les responsabilités (*black box*) identifie une suite d'essais fonctionnels pour les préoccupations. (2) Le test basé sur

les risques (*grey box*) utilise le modèle de fautes pour raffiner la suite d'essais de sorte que les défauts, dus au processus de ré-factorisation, comme la solution (orientée aspect), soient le plus susceptibles d'être capturés. Et finalement (3) un test de validation basé sur le code source de l'application sous test (*white box*) permettant d'inspecter les cas de test développés.

Par ailleurs, Zhou et al. [Zhou04] proposent une approche pour le test unitaire des aspects. Leur stratégie se divise en quatre phases. Le test des classes est effectué en premier afin d'éliminer les erreurs qui ne sont pas en relation avec les aspects. Chaque aspect est intégré et testé individuellement dans une seconde phase. La troisième étape consiste à intégrer et tester de manière incrémentale tous les aspects. Finalement, le système en entier est re-testé. Cette approche est basée sur le code source de l'application sous test. Dans le même ordre d'idée, Sereni [Sere03] propose une méthode de test pour l'analyse statique d'aspects basée sur un modèle syntaxique des indicateurs de pointcut en utilisant des expressions régulières.

Xie et al. [Xiet05] proposent un *framework* permettant la génération automatique de tests unitaires en utilisant le code compilé (*bytecode*) AspectJ. Dans le même contexte, Zhao [Zhao02] propose une approche basée sur les graphes de contrôle. Les classes et les aspects sont soumis à trois niveaux de test. Le test intra-module accomplit le test au niveau d'un module individuel tel qu'un advice, une introduction, ou une méthode d'aspect ou de classe. Le test inter-modules sert à tester un module public en relation avec les autres modules qu'il appelle (directement ou indirectement). Le troisième niveau de test vise à tester un ensemble de modules publics d'un aspect ou d'une classe lorsqu'ils sont appelés aléatoirement dans une séquence à l'extérieur de l'aspect ou de la classe. L'approche proposée par Zhao porte principalement sur le test unitaire des programmes aspects. Ces différentes approches sont plutôt basées sur le code source (aspects et classes) des applications sous test.

Par ailleurs, d'autres travaux se sont intéressés à la génération de séquences de test à partir des diagrammes d'états-transitions [Xud05-1, Xud04, Badr05]. Ces travaux ont

porté sur le comportement de classes auxquelles se greffe un aspect. Xu et al. [Xud05-2] proposent une approche orientée sur les modèles afin de produire des cas d'essais basés sur les interactions entre les aspects et les classes. Leurs modèles incluent les diagrammes de classes, les diagrammes d'aspects et les diagrammes de séquences.

Nous nous intéressons, dans ce mémoire, au comportement d'un groupe d'objets collaborant auxquels se greffent un ou plusieurs aspects. Une collaboration entre plusieurs objets permet de spécifier comment les objets interagissent dynamiquement pour la réalisation d'une tâche particulière. Le problème qui se pose, dans ce contexte, vient du fait que l'intégration des aspects peut altérer la collaboration des objets. Les aspects, de par leur nature, ont la possibilité de se greffer au contrôle et peuvent changer l'état du système [Alex04]. Nous devons donc nous assurer que les aspects s'intègrent correctement au contrôle de la collaboration. Les préoccupations implémentées à l'intérieur des aspects ont donc le potentiel d'étendre le comportement original d'une collaboration. La stratégie proposée, dans ce mémoire, représente une extension de la stratégie développée par Badri et al. [Badr04], basée sur les diagrammes de collaboration UML, pour les systèmes orientés objet.

CHAPITRE 3

LA TECHNOLOGIE ASPECT ET ASPECTJ ™

3.1 Motivations

La croissance de l'informatique ces dernières années a fait en sorte que la complexité des logiciels a augmenté radicalement. Il devenait donc inévitable de structurer le développement de ces programmes afin de faciliter le travail des développeurs. Alors sont venus les langages structurés permettant de décomposer les problèmes en sous procédures réduisant ainsi la complexité de chacune des tâches. Cependant, pendant que la complexité des systèmes augmentait, de meilleures techniques de programmation devenaient nécessaires. La programmation orientée objet (POO) a permis alors de considérer un système comme un ensemble de modules réalisant un travail commun. Les classes permettaient de cacher des détails d'exécution sous des interfaces. Le polymorphisme a fourni un comportement et une interface commune pour des concepts relatifs, et des composants plus spécialisés pour changer un comportement particulier sans avoir besoin de modifier les concepts de bas niveau. La principale force de la programmation orientée objet vient du fait qu'il est possible de modéliser les comportements semblables d'un objet à un seul endroit [Balt01] (les classes). Cette modélisation augmente grandement le niveau sémantique des systèmes complexes.

Cependant, même si la POO permet d'encapsuler les comportements communs à un même objet, certains problèmes sont toujours présents. Un développeur crée un logiciel souvent avec beaucoup de contraintes. Or, ces contraintes ont la possibilité de s'appliquer à plusieurs objets. La notion d'enchevêtrement des préoccupations devient donc un problème [Xiet05]. Bien que ce phénomène s'étende souvent au niveau de plusieurs modules, les implémentations actuelles tendent à mettre en application ces préoccupations en utilisant des méthodologies unidimensionnelles. Ceci se traduit par l'implémentation de ces fonctionnalités au sein même de toutes les classes les utilisant.

3.2 Symptômes

Quelques symptômes peuvent indiquer la présence de problèmes en relation avec l'enchevêtrement des préoccupations dans un système. Nous pouvons classer ces symptômes en deux catégories [Balt01] :

- *Enchevêtrement du code* : Les modules dans un système logiciel peuvent simultanément agir l'un sur l'autre avec plusieurs conditions. Par exemple, souvent les développeurs pensent simultanément à la logique, à l'exécution, à la synchronisation, à la notation et à la sécurité [Xiet05]. Une telle multitude de conditions a généralement comme conséquence un enchevêtrement du code.

- *Dispersion du code* : Sachant que les préoccupations, par définition, se retrouvent dans plusieurs modules, cela concerne également la duplication du code. Par exemple, dans un système utilisant une base de données, des soucis de synchronisation peuvent se retrouver à plusieurs endroits [Balt01].

L'enchevêtrement du code ainsi que sa dispersion ont des impacts majeurs sur la conception et le développement de systèmes. Ceci suppose une :

- *Mauvaise traçabilité* : Mettre en application simultanément plusieurs préoccupations rend difficile l'implémentation de ces préoccupations ainsi que leurs exécutions.

- *Diminution de la réutilisation du code* : Le fait qu'un module implémente plusieurs préoccupations diminue les chances qu'un autre système puisse utiliser ce même module [Asjw05]. Dans ce cas, le module en question utilise des fonctionnalités qui ne sont pas nécessaires en d'autres circonstances diminuant ainsi la cohésion à l'intérieur du module.

- *Faible qualité du code* : L'enchevêtrement du code peut produire des problèmes difficilement identifiables. Par exemple, en implémentant plusieurs préoccupations en même temps cela causera immanquablement que l'une des fonctionnalités échappera au contrôle du développeur.

- *Difficulté d'évolution* : Lorsqu'un système comporte trop de préoccupations transverses, les chances de le faire évoluer deviennent minces [Balt01]. Changer une préoccupation implique de la modifier dans tous les modules l'utilisant. En plus de demander un surcroît d'effort, de temps et d'argent, il faut s'assurer que ces changements sont appliqués partout. Cela implique également de re-tester tous les modules ayant subi un changement.

La figure 1 représente ce qui pourrait être un exemple de dispersion de diverses préoccupations transverses dans un système quelconque.

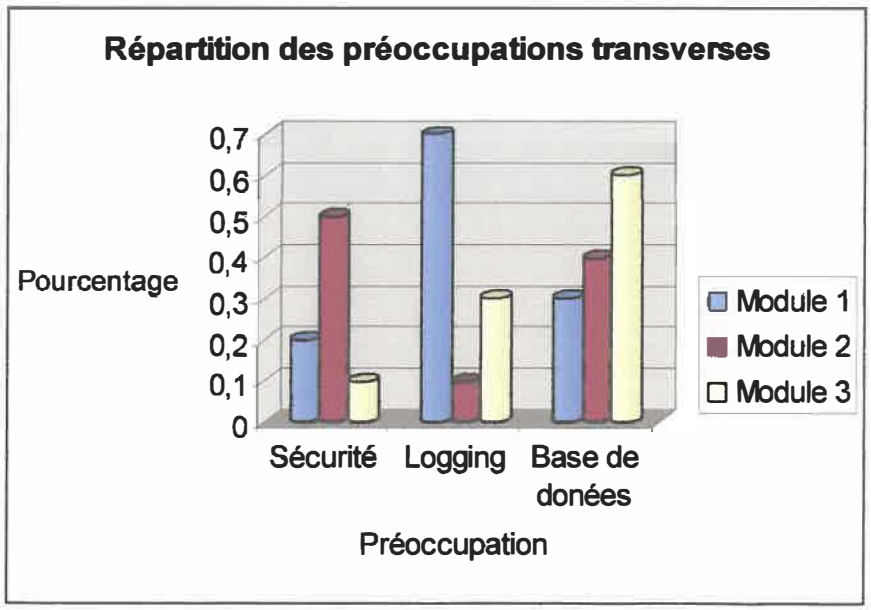


Figure 1 Répartition des préoccupations transverses dans un système

3.3 Une solution

La technologie aspect s'avère, à ce moment, un bon moyen pour combler ces problèmes. En regard avec les recherches effectuées dans ce domaine, nous pensons que le paradigme de programmation aspect sera la prochaine grande révolution dans le domaine du développement d'applications. L'utilisation de la technologie aspect permet principalement de :

- *Regrouper les préoccupations transverses* : La POA (programmation orientée aspect) permet de regrouper les préoccupations tout en minimisant le couplage avec les modules [Ajpg02]. Une réduction du code dupliqué est donc un avantage direct.

- *Réutiliser le code* : En éliminant les préoccupations transverses d'un module, nous augmentons les chances que ce module puisse être réutilisé dans un autre système [Balt01]. En général, une application comportant un niveau de couplage bas représente la clef pour la réutilisation du code. Toujours selon Baltus [Balt01], la POA permet la réalisation de systèmes plus faiblement couplés que la POO (programmation orientée objet).

3.4 AspectJ

AspectJ [Ajws05] représente une extension orientée aspect intéressante du langage Java. Il introduit plusieurs constructions relatives aux aspects telles que les introductions, les points de jointure, les points de coupure et les advice. Eclispe [Ajws05] offre un compilateur ainsi qu'une plateforme de développement pour les applications AspectJ. Un aspect rassemble des points de jointures et des advice pour former une unité de recoupement [Ajws05, Balt01]. Un aspect est similaire à une classe Java ou C++, dans le sens où, il contient des champs et des méthodes. Certaines constructions ont été ajoutées afin d'implémenter structurellement les préoccupations transverses. Les points de jointure ont comme principale tâche de définir la structure des préoccupations que l'on souhaite factoriser en désignant des

points bien précis dans l'exécution d'un programme. AspectJ permet de définir les points de jointure relatifs à un appel de méthode ou de constructeur [Balt01]. Il est possible de regrouper plusieurs points de jointure à l'aide des points de coupure. Un advice est un mécanisme (similaire à une méthode) utilisé pour spécifier le code à exécuter lors de la capture d'un point de jointure par le programme. L'avantage des advice vient du fait qu'ils peuvent avoir accès à certaines valeurs du contexte d'exécution d'un point de coupure. Les points de coupure et les advice définissent à eux deux, les règles d'intégration. Dans le but de familiariser le lecteur avec la technologie aspect, nous proposons une introduction au paradigme aspect dans ce qui suit. Ainsi, il sera plus facile de présenter au lecteur les détails de notre stratégie. Dans le cadre de nos travaux, nous avons utilisé la technologie AspectJ [Ajws05]. La section suivante présente les éléments essentiels de cette implémentation en relation avec des exemples concrets.

```
class  Point
{
    int x, y;
    public void setX(int x)
    {
        this.x = x;
    }
    public void setY(int x)
    {
        this.y = y;
    }
}
class  Line
{
    private Point p1, p2;
    public void setP1(Point p1)
    {
        this.p1 = p1;
    }
    public void setP2(Point p1)
    {
        this.p2 = p2;
    }
}
```

```
aspect PointBoundsPreCondition
{
    before(int x) : call(void Point.SetX(int)) && args(x)
    {
        if ( x < MIN_X || x > MAX_X)
        {
            throw new RuntimeException();
        }
    }
}
```

Figure 2 Exemple de code orienté aspect avec AspectJ

Il convient de mentionner qu'un aspect contient principalement des attributs et des méthodes comme une classe. Donc, essentiellement, nous pouvons étendre le comportement d'une classe à celui d'un aspect [Ajpg02, Balt01], c'est-à-dire :

1. Faire appel aux méthodes de l'aspect,
2. accéder aux attributs de l'aspect,
3. déclarer des méthodes,
4. etc.

Un aspect contient également certains mécanismes supplémentaires. La figure 2 présente un exemple d'aspect avec deux classes. Nous utiliserons cet exemple afin d'exposer au lecteur les concepts importants de la technologie aspect.

3.4.1 Les points de jointure

Le concept de points de jointure (*joint point*) est un élément essentiel dans le développement orienté aspect. Ceux-ci ont comme principale tâche de définir la structure des préoccupations que l'on désire factoriser. Les points de jointure désignent des points bien précis dans l'exécution d'un programme. AspectJ permet de définir les points de jointure à divers nivaux :

1. L'exécution d'une méthode ou d'un constructeur (on se trouve dans le contexte de l'exécution, donc de l'appelé).
2. L'accès en lecture ou écriture d'un champ.

Tableau I

Types de base des advice pour AspectJ

Type	Utilisation
Before	S'exécute avant un point de jointure ou un point de coupure.
After	S'exécute après un point de jointure ou un point de coupure.
Around	Remplace complètement un point de jointure ou un point de coupure.

3.4.4 Les aspects

Un aspect rassemble des pointcut et des advice pour former une unité de recoupement [Ajws05, Balt01]. Les points de coupure et les advice définissent à eux deux les règles d'intégration. Un aspect est similaire à une classe Java ou C++, dans ce sens où, il contient des champs et des méthodes. Pratiquement identique à une classe, la déclaration de l'aspect se fera en substituant le mot clef *class* par *aspect* [Balt01]. L'aspect *PointBoundsPrecondition* est défini de la ligne 25 à 34 à la figure 2.

La figure 3 donne sous forme graphique la représentation des concepts présentés plus haut. Nous remarquons que l'instruction 25 correspond au point d'entrée de l'aspect. Étant donné que cet aspect possède uniquement un point de jointure, aucun point de coupure n'a besoin d'être défini (c'est un point de coupure *anonyme*). Cela dit, le point de jointure correspond à la ligne 27 et fait référence à la méthode *setX* de la classe *Point*. Quant à l'advice, il est du type *before* ce qui nous indique que les opérations seront effectuées avant l'appel de la méthode *setX*. Le corps de l'advice consiste uniquement à vérifier si le paramètre *x* est compris entre les bornes *MIN_X* et *MAX_X*.

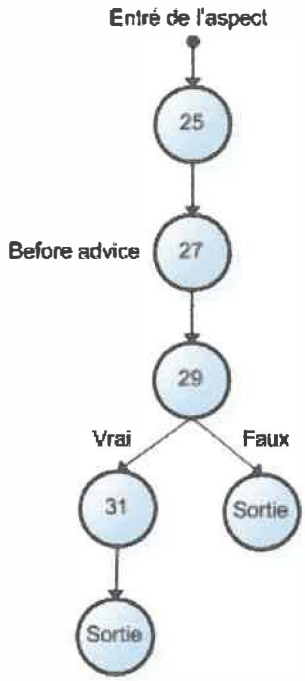


Figure 3 Arbre d'exécution de l'aspect *PointBoundsPrecondition*

Il est maintenant évident que le développement orienté aspect rajoute un niveau d'abstraction supplémentaire en termes de contrôle. Nous croyons que les méthodes de test orienté objet ne pourraient pas être complètement adéquates dans ce genre de contexte. Ceci vient principalement du fait que les méthodes de test orienté objet s'intéressent à la classe en termes de test unitaire. Les aspects étant invisibles au niveau des classes, il devient difficile d'intercepter les nouveaux messages introduits.

CHAPITRE 4

MÉTHODOLOGIE DE L'APPROCHE

La méthodologie proposée se divise en deux principales étapes (figure 4). La première porte sur la génération des séquences de test. Chaque séquence de test générée correspond à un scénario particulier du diagramme de collaboration. Les séquences générées couvrent le scénario de base (*happy path*) [Larm03] ainsi que ses différentes extensions. Les diagrammes de collaboration ainsi que les aspects sont, dans le contexte de notre approche, décrits formellement en utilisant XML. La démarche adoptée consiste à générer, dans un premier temps, les séquences de test relatives aux différents scénarios de la collaboration sans les aspects. Cette étape vise à tester séparément la collaboration. Ceci nous permet de nous assurer dans un premier temps que les objets (sans les aspects) collaborent correctement pour la réalisation de la tâche visée et d'éliminer par conséquent les erreurs qui ne sont pas en relation avec les aspects [Mass05-1]. L'intégration des aspects se fait dans un second temps itérativement selon les critères définis au chapitre 7. Cette démarche nous permet, en fonction des aspects, de déterminer leur impact sur les scénarios de la collaboration originale et d'identifier formellement les séquences (scénarios) auxquelles les aspects viennent se greffer. Les messages introduits par les aspects aux classes sont intégrés automatiquement aux séquences de base et sont testés lors du processus de vérification. Finalement, tous les aspects sont intégrés au diagramme de collaboration afin de s'assurer que le comportement original n'est pas altéré.

Cette approche nous permet, en fait, de réduire la complexité du test due à l'intégration des aspects aux classes collaborantes. La seconde étape consiste, quant à elle, à supporter automatiquement le processus de vérification de l'exécution des séquences de test (conformité par rapport aux spécifications). Cette étape nécessite une instrumentation du code source de l'application à tester. Nous utilisons, lors de cette étape, un aspect pour capter les appels des méthodes dans le code de l'application à tester. Cette étape permet de s'assurer de la conformité du code à la spécification décrite dans le diagramme de collaboration. Les séquences complètes

obtenues après intégration des aspects tiennent donc compte à la fois de la collaboration originale entre les objets, du contrôle lié à cette collaboration, des pré et post conditions reliées aux interactions entre objets ainsi que des aspects. Ceci nous permet de supporter formellement le processus de vérification.

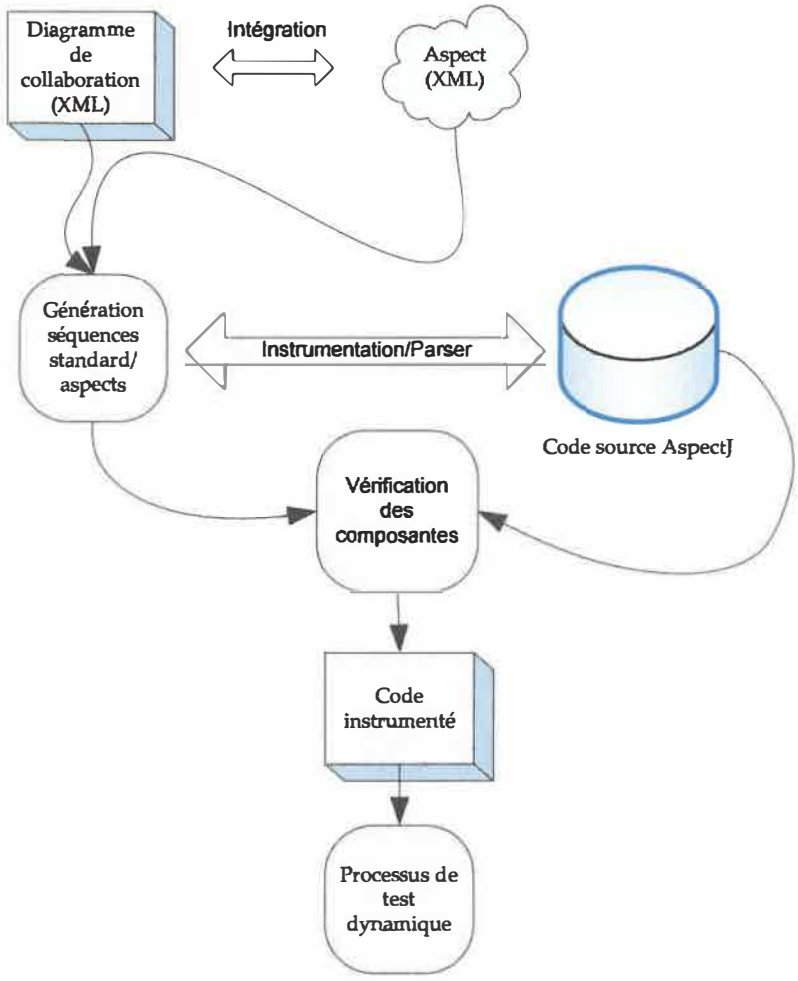


Figure 4 Modèle général de notre stratégie de test.

CHAPITRE 5

CHALLENGES

Le développement orienté aspect facilite grandement la tâche des développeurs [Aswb05]. Cependant, ce nouveau paradigme de programmation impose de nouveaux challenges en termes d'analyse et de test. La principale problématique vient de la relation caractéristique entre les aspects et les classes, qui est différente de la relation classe/classe dans les systèmes orientés objet. Les liens unissant un aspect à une classe ne sont pas identifiables en analysant les classes [Alex04, Mort04, Zhou04]. Les aspects ont également la possibilité de causer certains conflits en changeant le contrôle de l'exécution inadéquatement [Mort04] lorsque les points de jointure sont trop faibles ou trop forts. De plus, un aspect à la possibilité d'affecter le comportement de plusieurs classes.

Une des formes majeures de dépendances entre les aspects et les classes vient du fait que le concept de l'appelant et de l'appelé prend un nouveau sens [Balt01]. La plupart des stratégies de test orienté objet se basent sur ce genre de relation entre les classes [Ball98]. Premièrement, dans un système traditionnel un appelant spécifie les différents appels qu'il effectuera, et quand ils seront effectués. Par exemple, il est possible qu'une classe utilise diverses méthodes d'une autre classe. Dans un système orienté aspect l'inverse se produit puisque les règles d'intégrations sont définies dans les aspects à l'insu des classes. L'aspect décrit à l'aide de diverses constructions comment cet appel sera effectué. Autrement dit, les classes ignorent totalement l'existence des aspects dans un système.

Dans un système orienté objet, l'appelant et l'appelé, à un haut niveau, sont des classes. Dans un système orienté aspect, ces participants incluent maintenant les aspects, ce qui est un concept différent en soi. En raison de la similitude et de la différence entre le rapport aspect/classes et classe/classe, il n'est pas possible d'utiliser les méthodes de test OO intégralement. Toutefois, il est envisageable de

les adapter afin de tenir compte de ces différences. La figure 5 schématise les diverses relations possibles dans un système objet et aspect.

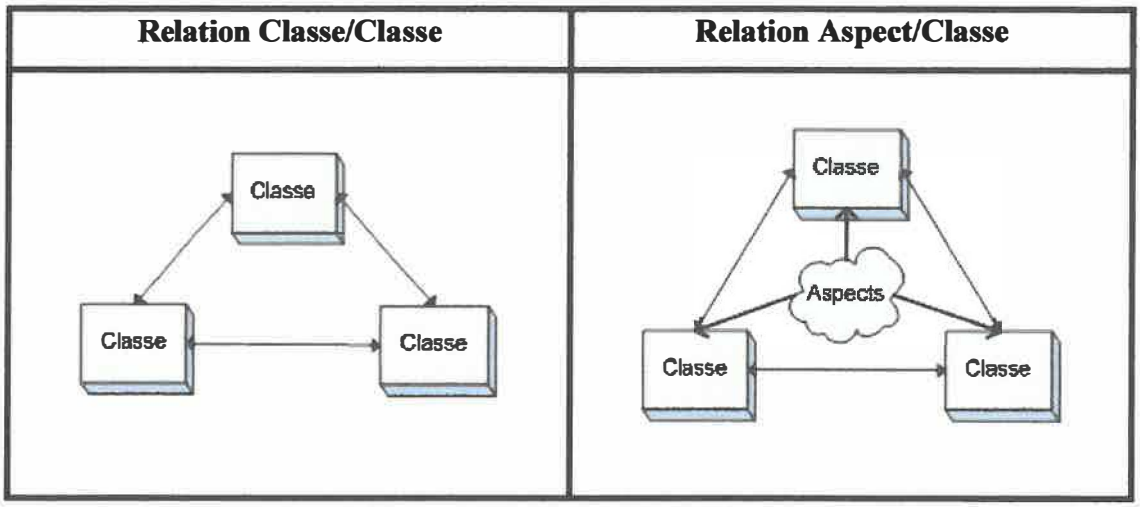


Figure 5. Type de relation en OO vs. OA

Selon Alexander et al. [Alex04], l'orienté aspect supporte les séparations des préoccupations et apporte des niveaux de dépendances explicites et implicites (appelant/appelé). Ainsi, les systèmes développés à l'aide de la technologie aspect apportent de nouveaux challenges en termes de test. Cela inclut de nouvelles sources de fautes au moment de l'exécution. Il pose également certaines questions intéressantes : « *Comment testons-nous adéquatement les systèmes orientés aspect ?* » ou « *Comment savons-nous que nos tests ainsi que les objectifs de qualité ont été atteints ?* ». Ce genre de questionnement est très pertinent, car ils permettent de poser les bases de la problématique générale du test orienté aspect. Ces questions ne sont toutefois pas faciles à répondre, car :

Les interactions entre les méthodes et les advice

Les interactions entre les méthodes de classes et les advice se font principalement à l'aide des points de jointure. Cela dit, aucun mécanisme ne permet de tester adéquatement ces relations. En d'autres termes, chaque préoccupation doit être testée dans le bon contexte d'exécution.

Les aspects n'ont pas d'identité propre

En effet, ils dépendent du contexte des autres classes en ce qui concerne leur identité et aussi de leur contexte d'exécution. Un aspect permet de factoriser les préoccupations transverses à plusieurs classes et ne peut donc pas exister seul.

L'implémentation des aspects est grandement liée au contexte de greffe

Les aspects dépendent de l'implémentation interne des classes auxquelles ils sont reliés. Toutes les modifications effectuées à l'intérieur de ces classes ont des chances d'affecter ces liens. Ces changements ont la possibilité d'interrompre l'exécution des aspects et ainsi modifier les séquences initialement prévues. Par exemple, si nous modifions la signature d'une méthode de classe (changement dans le nom de la fonction, des paramètres, etc.), il est possible que l'appel d'un advice ne s'effectue pas au bon endroit. Il est envisageable que l'ajout d'une nouvelle méthode de classe corresponde aux règles d'intégration définies dans certains aspects. Ainsi, les advice de ces aspects seront exécutés à des endroits non voulus.

Le contrôle ainsi que les dépendances ne sont pas toujours visibles facilement

Dû à la nature du processus de greffe, les développeurs ne peuvent pas toujours prévoir le contrôle d'exécution. Ceci a pour conséquence de rendre difficile l'identification des erreurs. De nouveaux outils doivent être développés afin de visualiser facilement les dépendances entre les aspects et les classes.

L'ordonnancement

Aucun mécanisme ne permet de s'assurer de l'ordonnancement lorsque plusieurs aspects (en cas de conflit avec les advice) se greffent à une classe. Ainsi, des effets de bord sont toujours possibles. Un problème d'ordonnancement fera en sorte que plusieurs scénarios aléatoires pourront être exécutés. Il est donc préférable dans ce

cas de s'assurer que tous les cas possibles n'introduiront pas de fautes dans le système.

Afin de fournir une piste dans le développement de méthodes de test pour le paradigme aspect, Alexander [Alex04] propose d'analyser les types de fautes pouvant survenir dans un système aspect. Lorsqu'une erreur survient, il est important de diagnostiquer cette erreur et d'en détecter l'origine. Suite à cela, il devient possible d'en déterminer la cause et ainsi corriger la situation. L'instrumentation du code objet peut fournir de très bons outils dans la découverte de ces erreurs. Cependant, pour détecter toutes les possibilités de fautes dans un système orienté aspect, nous devons aussi examiner le code des aspects qui sera greffé au corps des classes ciblées. Quatre sources potentielles peuvent survenir avec les aspects :

1. La faute réside dans une portion du corps qui n'est pas affectée par l'aspect

Ce type de faute peut survenir même si aucun aspect ne s'intègre au corps des classes. Dans cette situation, la faute n'est strictement pas liée à l'utilisation de la technologie aspect. Différentes méthodes (également stratégies de test) de test orienté objet ont déjà été proposées afin de couvrir ce genre d'erreur. C'est pourquoi nous devons avoir un niveau de confiance suffisamment élevé en notre logiciel afin de ne pas confondre ces erreurs avec celles provoquées par les aspects.

2. La faute réside dans une portion du code spécifique à l'aspect qui n'est pas en relation avec le contexte d'exécution.

Dans ce cas, la faute est présente dans l'aspect et surviendrait, peu importe la composition incluant l'aspect. En d'autres termes, ce type de faute est indépendant du contrôle lors du processus de greffe.

3. La faute émerge de l'interaction entre un aspect et une classe.

Souvent, lors du processus de greffe de code, de nouvelles dépendances sont introduites. Ces dépendances ajoutent des contraintes supplémentaires pouvant causer des fautes qui sont uniquement envisageables lors d'interactions entre aspects et classes. Plus un point de jointure sera flexible, plus il y aura d'interactions avec les classes d'un système. Du même coup, plus il y a des chances que des erreurs surgissent.

4. La faute survient lorsque plus d'un aspect se greffent à une classe.

Pour les systèmes contenant plusieurs aspects, il est possible que certains conflits surviennent lorsque plus d'un advice se greffent à un même endroit. Même si AspectJ [Ajws05] définit certaines constructions pour l'ordonnancement, il est toujours possible que deux advice soient aléatoirement exécutés [Mort04]. Ce type de faute peut survenir seulement lorsqu'une certaine permutation entre les aspects et la classe est exécutée.

La technologie aspect voulant s'intégrer de plus en plus au développement de logiciels d'envergure [Ajpg02], nous croyons qu'il est nécessaire d'étendre les mécanismes de test orienté objet. Notre méthode vise à couvrir les niveaux d'abstractions supplémentaires apportées par la technologie aspect présentées dans ce chapitre. Ces notions ne sont pas couvertes par les techniques de test standard puisque la relation caractéristique entre un aspect et une classe est différente de celle entre deux classes dans un système orienté objet. Les aspects définissent eux-mêmes les règles d'intégration aux classes. L'appelant et l'appelé deviennent donc la même entité [Zhou04]. Dans ce contexte, il est impossible de vérifier si l'intégration de l'aspect à la classe s'effectue correctement.

CHAPITRE 6

DIAGRAMMES DE COLLABORATION

Dans les systèmes orientés objet, les objets interagissent pour implémenter le comportement dynamique. Le concept orienté objet traite, en particulier, de la définition des objets et de leurs collaborations [Offu99-1, Offu96]. Le comportement peut être décrit essentiellement à deux niveaux. Le premier niveau porte sur le comportement individuel des objets tandis que le second porte sur leur comportement de groupe [Abdu00]. Le comportement collectif d'un groupe d'objets, pour la réalisation d'une tâche donnée, peut être spécifié à l'aide des diagrammes de collaboration (interaction) UML [Rrsc98]. Les diagrammes de collaboration UML permettent de décrire les relations de dépendances entre les interactions [Zhou04]. Ils illustrent les interactions entre les objets sous forme de graphes ou de réseaux. Chaque scénario correspond à une séquence bien précise dans la collaboration.

C. Larman [Larm03] mentionne que la principale force des diagrammes de collaboration est d'illustrer efficacement des branchements complexes, des itérations et des comportements concurrents. Chaque scénario correspond à une séquence bien précise dans le diagramme de collaboration. Les liens unissant les classes impliquées dans la réalisation d'un scénario représentent les opérations. Une opération est la spécification d'une transformation (requête) qu'un objet peut être appelé à faire. Elle a un nom et une liste de paramètres. Une méthode est un procédé qui met en application une opération. Une opération possède généralement un algorithme [Ajpg02]. Les diagrammes de collaboration fournissent les six informations suivantes :

1. Les objets qui sont utilisés au sein des interactions ainsi que la structure de ces objets.
2. Les instances permises des séquences d'opérations d'un objet.
3. La sémantique d'une opération.

4. Les opérations qui sont importées d'une classe, qui permettent une collaboration avec des objets d'une autre classe.
5. Le modèle de communication des objets en collaboration (synchrone ou asynchrone).
6. Les caractéristiques d'exécution des objets (parallèle ou séquentiel).

Nous nous intéressons, dans le cadre de notre approche, à l'impact de l'intégration d'un ou de plusieurs aspects à un groupe de classes collaborantes. Selon le modèle de faute présenté par Alexander et al. [Alex04], deux situations seraient susceptibles d'être à l'origine des défaillances. La première est en relation avec le lien unissant l'aspect et l'abstraction primaire qui survient lorsque la greffe introduit de nouvelles dépendances. La deuxième concerne le cas où plusieurs aspects se greffent à une classe. Dans ce cas, il devient difficile de localiser l'origine d'une erreur. Les diverses permutations en termes de contrôle entre les aspects et la classe peuvent rendre difficile la localisation de la source de l'erreur. Pour réduire cette complexité, nous avons adopté une approche itérative pour l'intégration des aspects. Les critères introduits dans le chapitre qui suit ont pour objectif de couvrir les nouvelles dimensions introduites par l'intégration des aspects à un groupe de classes collaborantes. Les préoccupations soutenues par les aspects ont la possibilité d'apporter certains changements en termes de contrôle [Zhou04], ce qui doit être évalué et testé adéquatement.

CHAPITRE 7

CRITÈRES DE TEST

Un critère de test est une règle ou un ensemble de règles imposant des conditions sur des stratégies de test [Offu99-2, Xiet05, Abdu00]. Il spécifie également les tests requis en termes de dispositifs identifiables de la spécification logicielle en évaluant une série de cas de test (aussi connu *suite de test*) [Offu99-1]. Les critères de test sont utilisés afin de déterminer ce qui doit être testé sans toutefois préciser comment le tester. Les ingénieurs de test se basent sur les critères de test pour mesurer les couvertures de test [Wuye02]. Ils permettent également d'évaluer la qualité d'une suite de tests. Les deux premiers critères présentés tiennent compte des diagrammes de collaboration entre objets [Abdu00, Badr04, Xiet05, Zhou04]. Nous complétons ces critères par des critères tenant compte des nouvelles dimensions introduites par l'intégration des aspects.

Le diagramme de collaboration présenté à la figure 6 servira d'exemple afin d'illustrer plus clairement les deux critères de test proposés par J. Offutt [Offu99-2].

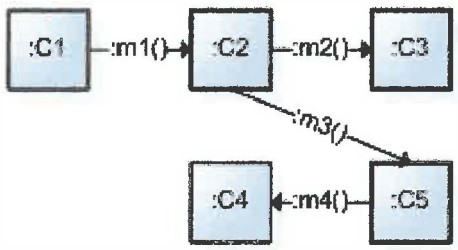


Figure 6 Diagramme de collaboration générique.

7.1 Critère de transition

Dans un diagramme de collaboration, une transition représente une interaction entre deux objets. Ce passage s'effectue lorsqu'un stimulus extérieur provoque le déclenchement d'un message impliqué dans le diagramme de collaboration. Chaque transition doit être testée au moins une fois [Offu99-2]. Selon Xie [Xiet05], un

testeur devrait également tester chaque pré condition de la spécification au moins une fois afin de s'assurer qu'il sera toujours possible d'exécuter un scénario donné (d'une pré condition mal établie résultera un scénario qui ne sera jamais exécuté). Un test traverse une transition uniquement lorsque la pré condition correspondante est vraie.

C1: Chaque message dans un diagramme de collaboration doit être testé au moins une fois.

Dans ce cas, quatre interactions sont possibles (Tableau II).

Tableau II

Transitions possibles du diagramme de collaboration.

Transition	Message
:C1 → :C2	*m1*
:C2 → :C3	*m2*
:C2 → :C5	*m3*
:C5 → :C4	*m4*

7.2 Critère de séquence

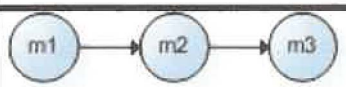
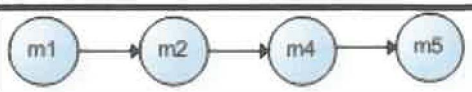
Le critère précédent porte sur le test des transitions prises individuellement. Il ne couvre pas les séquences de transitions. Une séquence correspond à une suite logique de plusieurs interactions. Ceci représente en réalité un scénario bien précis de la collaboration ayant la possibilité d'être exécuté au moins une fois lors du déroulement du programme. En testant les séquences avec les différents éléments de contrôle (pré et post condition en particulier), nous vérifions toutes les possibilités de la collaboration (scénario de base avec ses extensions). Dans certains cas, le nombre de séquences est illimité (présence d'itérations). Il revient au testeur de sélectionner les séquences les plus représentatives.

C2: Chaque séquence valide dans le diagramme de collaboration doit être testée au moins une fois.

Toujours en se basant sur le diagramme de collaboration présenté à la figure 6, nous constatons qu'il existe deux scénarios (donc deux séquences) possibles comme décrits dans le tableau III.

Tableau III

Séquences de base du diagramme de collaboration.

Numéro de séquence	**Séquence**
1	
2	

L'importance de ces critères de test se situe à deux niveaux. D'une part, parce que le critère concernant les interactions s'intéresse uniquement aux messages de manière individuelle. D'autre part, nous testons tous les scénarios possibles. Nous remarquons que le critère *C2* inclut le critère *C1*. Cependant, il faut faire une distinction importante, même si le test des transitions n'a présenté aucun problème rien ne garantit que l'exécution des différents scénarios s'enchaînera correctement. Il est toujours possible qu'une suite bien précise de certains messages provoque des erreurs. Cependant, si nous avons testé toutes les transitions dans un premier temps, nous pouvons conclure que l'erreur provient du contexte. À ce moment, nous pouvons déterminer plus facilement l'origine de l'erreur sachant que chacun des messages a été vérifié au départ.

Ces deux critères ont démontré l'importance qu'ils jouaient afin de déterminer ce qui doit être testé dans un diagramme de collaboration de base. Cependant, dans notre cas, nos diagrammes de collaboration incluent des aspects. Comme nous l'avons

démontré à la section 5, les aspects introduisent de nouvelles dépendances qui sont à toute fin invisible au niveau des classes. Les critères de test présentés à cette section ne couvrent pas les différents éléments apportés par l'intégration des aspects au diagramme de collaboration. Principalement, car ceux-ci s'intéressent uniquement aux relations du type classe/classe. Les nouveaux éléments apportés par les aspects au diagramme de collaboration ne sont pas visibles au niveau du test des transitions et des séquences. Un aspect introduira de nouveaux messages dans la réalisation d'un cas d'utilisation qui devront être pris en compte. Certes, les critères de test présentés à cette section sont toujours pertinents pour nous.

7.3 Critère de couverture des séquences modifiées

Dans un premier lieu, la collaboration entre les objets est testée sans les aspects afin de s'assurer que les différents scénarios sont implémentés correctement. Ceci est possible puisqu'un aspect, de par sa nature, ne doit pas modifier la sémantique d'une classe [Zhou04]. Dans le cas contraire, il devient préférable de créer un nouvel objet ayant une identité propre. Les aspects dépendent du contexte des autres classes en ce qui concerne leur identité, comme mentionné par Alexander et al [Alex04]. Les aspects sont donc en relation directe avec ces classes et ne peuvent exister seuls. Subséquemment, une classe devrait être en mesure d'accomplir son rôle en fonction des responsabilités originales qui lui ont été affectées lors de sa conception. Cependant, les aspects introduisent de nouveaux messages qui s'intègrent à la collaboration. Ces messages ont la possibilité, dans une certaine mesure, de modifier l'état du système [Mort04] et d'altérer la collaboration. Il est donc impératif de tester adéquatement les séquences affectées par les aspects.

C3: Toutes les séquences de la collaboration modifiées par un ou plusieurs aspects doivent être re-testées.

7.4 Critère d'intégration simple

Lorsqu'un seul aspect se greffe à une classe, nous parlons d'intégration simple. Nous devons donc identifier les séquences de la collaboration affectées par cette intégration et les re-tester. Ce type de test vérifie principalement les liens unissant les aspects aux classes sans tenir compte du contexte. La figure 7 présente le concept de l'intégration simple.

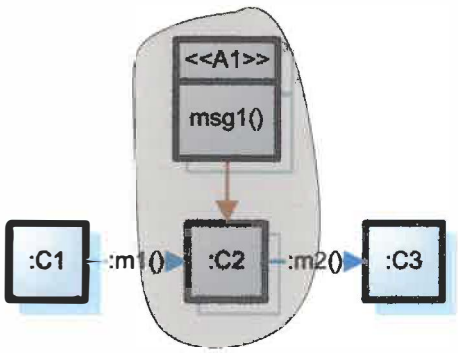


Figure 7 Intégration simple d'un aspect au sein d'un diagramme de collaboration.

C4: Si une méthode de classe est affectée par un advice et que cette méthode est utilisée dans le diagramme de collaboration à tester, toutes les séquences utilisant cette méthode doivent être re-testées.

7.5 Critère d'intégration multi-aspects

Dans un second lieu, nous étudierons le comportement d'une classe lorsque plusieurs aspects viennent s'y greffer. Il est à noter que nous présenterons le cas le plus simple c'est-à-dire avec deux aspects. Il est important de comprendre que ce phénomène se généralise très facilement pour des intégrations contenant plusieurs aspects.

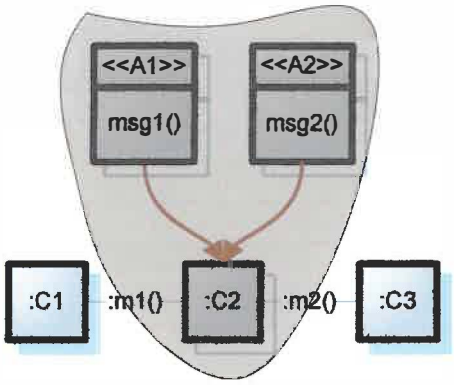


Figure 8 Intégration multi-aspects au sein d'un diagramme de collaboration.

Un problème éloquent du paradigme aspect, est qu'il précise difficilement les règles d'ordonnancement lorsque plusieurs advice se greffent à un même endroit dans un système (figure 8). Ceci peut causer certains conflits puisqu'il est impossible de prévoir le comportement de cette intégration. En d'autres termes, il est difficile de connaître la séquence d'exécution qui sera réalisée. Il est tout aussi ardu de tester un comportement aléatoire. Pour contrer ce problème, il convient de décortiquer tous les chemins envisageables suite à l'introduction de plusieurs aspects. Si les deux advice sont du type *before* nous aurons les deux possibilités suivantes (tableau IV) :

Tableau IV

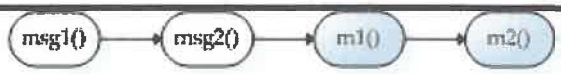
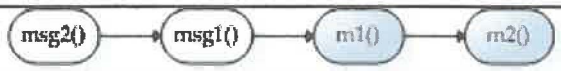
Présentation d'un cas de permutation possible.

Permutation 1	msg1() → msg2() → m1()
Permutation 2	msg2() → msg1() → m1()

Dans ce contexte, le choix de la séquence qui s'exécutera se fait aléatoirement par le compilateur. Nous devons tenir compte de toutes les éventualités qui pourraient survenir. Ce qui amène à considérer deux séquences présentées à l'aide du tableau V.

Tableau V

Séquences alternatives en relation avec une permutation.

Séquence 1	
Séquence 2	

Puisqu'il existe plusieurs types d'advice, nous devons être en mesure de faire face à toutes les éventualités possibles. Pour ce faire, il suffit de prendre tous les aspects s'intégrant à la classe et insérer les nouveaux messages aux endroits correspondants. Lorsque plusieurs advice s'intègrent à la même place, il faut créer les nouvelles séquences en utilisant toutes les permutations possibles. Ces conflits surviennent lorsque :

1. Plus d'un advice de type *before* s'intègrent avant un message de classe.
2. Plus d'un advice de type *after* s'intègrent après un message de classe.
3. Plus d'un advice de type *around* s'intègrent à un message de classe.

Le critère d'intégration multi-aspects se formule comme suit :

C5: Si une méthode de classe est affectée par plusieurs advice et que cette méthode est utilisée dans le diagramme de collaboration à tester, les séquences incluant cette méthode doivent être re-testées. Ce test devra être effectué en tenant compte de toutes les permutations possibles apportées par l'intégration de ces advice.

CHAPITRE 8

INTÉGRATION DES ASPECTS AUX CLASSES

Quels sont les impacts suite à l'intégration d'un ou de plusieurs aspects au niveau d'un diagramme de collaboration ? Nous croyons que cette intégration peut s'effectuer de plusieurs manières. Comment procéder au test ? Comment définir un modèle précis d'intégration afin de fournir une couverture de test complète et efficace ? Deux stratégies semblent à notre avis envisageables.

8.1 L'intégration complète

Dans la plupart des systèmes étudiés, nous avons pu constater que l'utilisation des aspects était plutôt limitée. Il devrait donc être possible d'intégrer tous les aspects aux diagrammes de collaboration et de tester l'ensemble de cette intégration en une seule itération. Certains auteurs [Beiz90] parlent d'une intégration « *big-bang* » qui est également utilisée au niveau du test orienté objet. Nous pouvons résumer cette approche de la façon suivante : « exécute et regarde ce qui se passe ». Toutefois, la plupart des auteurs ne préconisent pas ce type d'approche. La principale raison est que si une erreur survient, il sera plus difficile de localiser l'origine de la faute. Le même principe peut s'appliquer pour les systèmes orientés aspect. Les aspects sont intégrés au diagramme de collaboration de manière non formelle (aucun ordre précis, aucune règle d'intégration particulière) et ensuite testés afin de s'assurer que tout fonctionne correctement.

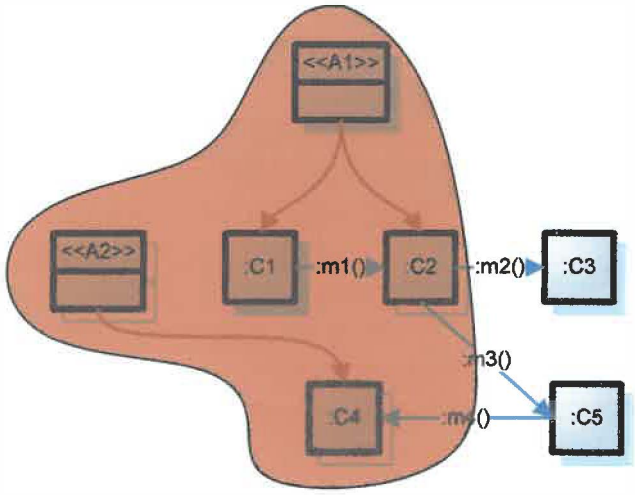


Figure 9 Intégration complète d'un ensemble d'aspects.

La figure 9 nous indique que deux aspects *A1* et *A2* se greffent respectivement aux classes *C1*, *C2* et *C4*. Toutes les séquences seront créées en intégrant tous les nouveaux messages apportés par les aspects.

Cependant, ce type d'approche cause certains problèmes. Principalement lorsque :

1. Plusieurs aspects sont dans le diagramme de collaboration.
2. Plusieurs aspects se greffent à une même classe.

Si l'un de ces deux scénarios se présente, il peut être difficile de déterminer l'origine de l'erreur en cas de défaillance. En résumé, ce procédé est coûteux en terme de ressources et si un échec survient nous ne pouvons pas identifier facilement l'élément défectueux. De plus, la non découverte d'erreurs ne garantit en rien qu'il n'en n'existe pas [Beiz90]. Plus le nombre d'aspects à intégrer est important plus il sera difficile de tester en une seule itération le diagramme de collaboration en question. Dans le même ordre d'idée, les critères de test présentés au chapitre 7 sont en contradiction avec l'essence même de ce type d'intégration. Pour ces raisons, nous croyons que cette approche n'est pas efficace. Nous croyons que les aspects doivent être intégrés de manière incrémentale au diagramme de collaboration.

8.2 Intégration incrémentale

Un aspect de par sa nature ne doit pas modifier la sémantique d'une classe [Alex04]. Dans le cas contraire, il devient préférable de créer un nouvel objet ayant une identité propre. Selon Ball [Ball98], les aspects dépendent du contexte des classes en ce qui concerne leur identité. Les aspects sont donc en relation directe avec ces classes et ne peuvent exister seuls. Subséquemment, une classe devrait être en mesure d'accomplir son rôle en fonction des responsabilités originales qui lui ont été affectées lors de sa conception. Dans le même ordre d'idée, un groupe d'objets exécutant une tâche commune devrait donc être capable d'effectuer un même travail avec ou sans l'utilisation d'aspects. Yuewei Zhou [Zhou04] détermine que chaque aspect devrait gérer uniquement une seule préoccupation. Le fait d'avoir plusieurs besoins s'entrecoupant dans un seul aspect n'est pas souhaitable puisqu'il est en contradiction avec l'essence même de l'approche orientée aspect [Alex04]. Par exemple, un aspect gérant la gestion du *logging* et de certaines politiques de sécurité rendrait la maintenance difficile. Donc, plus le nombre de préoccupations augmente dans un aspect, plus le degré de cohésion diminue. Cet auteur détermine également que ces préoccupations sont généralement indépendantes l'une de l'autre. L'utilisation d'un aspect ne devrait pas avoir d'impact sur un autre.

Pour ces raisons, il est plus judicieux de vouloir tester d'abord la collaboration (réalisation d'une tâche donnée) sans l'intégration des préoccupations. Le but de cette approche est d'isoler et d'éliminer les erreurs qui ne sont pas en relation avec les aspects. Si nous pouvons, par cette étape, atteindre un haut niveau de confiance en nos classes (aucune erreur), nous aurons de meilleures chances de découvrir les erreurs relatives aux aspects par la suite. Cette démarche a pour but d'épurer, dans un premier temps, le diagramme permettant ainsi de se concentrer sur la fonctionnalité à tester. La vérification de la collaboration des objets se voit facilitée par la même occasion. Si une erreur survient à cette première étape, nous savons qu'il s'agit d'un problème lié au cas d'utilisation (collaboration entre les objets de base) et non à l'utilisation des aspects. Ceci soutient également l'affirmation d'Alexander et al. [Alex04] disant qu'il est possible qu'une erreur survienne dans

une portion de code qui n'est pas affectée par les aspects. Pour des raisons de simplicité, nous proposons d'effectuer l'intégration des aspects de manière incrémentale. C'est-à-dire intégrer un aspect à la fois. Lorsqu'un aspect est intégré au diagramme, nous re-testons. Si une erreur survient à ce moment, nous pouvons conclure que cette erreur est issue de la relation entre l'aspect et la classe en question. Si par contre aucune erreur n'émerge de ce test, nous procédons à la prochaine intégration et nous testons de nouveau. Ainsi de suite jusqu'à ce que tous les aspects soient intégrés. Finalement, le diagramme de collaboration est testé dans son ensemble. Cependant, comment déterminer cet ordre d'intégration ? Yuewei Zhou [Zhou04] affirme qu'aucune importance ne devrait être accordée à l'ordonnancement. Il précise que peu importe l'ordre d'intégration choisi, le résultat sera le même. Notre point de vue diffère de celui proposé par cet auteur. Nous croyons qu'il serait préférable de déterminer un ordre précis d'intégration. Dans le but de faciliter la détection d'erreur, nous intégrerons les aspects en commençant par le plus complexe pour finir par le plus simple.

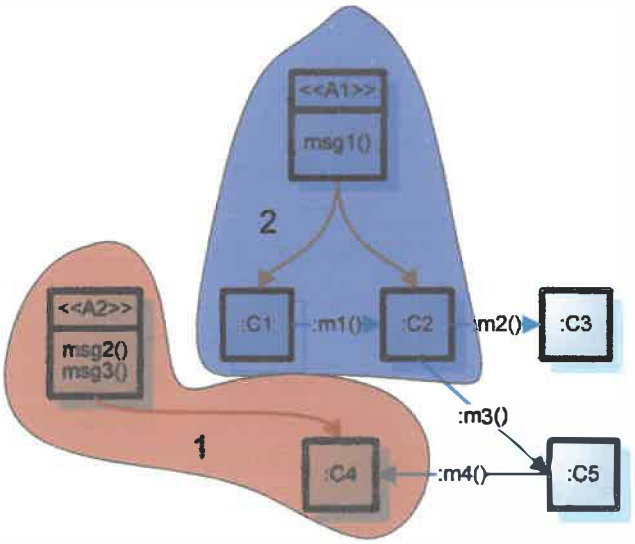


Figure 10 Intégration incrémentale de plusieurs aspects au niveau d'un diagramme de collaboration.

Selon la figure 10, l'aspect *A2* introduit deux méthodes *msg2()* et *msg3()* à la classe *C4*. Dans ce cas-ci, ces deux méthodes se greffent à la méthode *m4()*. Nous constatons ainsi que cet aspect est plus complexe que l'aspect *A1*. Il convient de

tester les séquences touchées par l'introduction de l'aspect *A2* dans un premier temps et par la suite celles touchées par la greffe de l'aspect *A1*. Sachant que dans une première étape le diagramme de collaboration a été testé dans le but de vérifier si les scénarios sont exempts d'erreurs, il devient donc inutile de tout re-tester. Il s'agit à ce moment de vérifier les séquences qui ont été modifiées et ainsi re-tester uniquement celles qui diffèrent des séquences originales. Si le diagramme de collaboration propose plusieurs scénarios, nous re-testons uniquement ceux touchés par le ou les aspects.

CHAPITRE 9

GÉNÉRATION DES SÉQUENCES DE TEST

9.1 Approche

Le processus de génération de séquences de test prend en compte les divers éléments de contrôle décrits dans le diagramme de collaboration. Chaque séquence valide du diagramme de collaboration correspond à un scénario d'exécution possible. Les séquences générées intègrent également les différentes interactions entre les aspects et les classes collaborantes. Notre démarche prend donc en considération aussi bien le niveau d'intégration classe/classe qu'aspect/classe. L'approche adoptée permet de tester de façon incrémentale la collaboration d'un groupe de classes auxquelles se greffent plusieurs aspects [Mass05-1, Mass05-2].

Pour présenter l'approche, nous allons utiliser un exemple réel provenant du site web AspectJ [Ajws05]. Ce diagramme utilise sept classes ainsi que deux aspects. Il représente un système d'appels comportant un modèle simple des raccordements de téléphone auxquels la synchronisation et les dispositifs de facturation sont ajoutés en utilisant des aspects. Nous avons modélisé ces deux scénarios à l'aide du diagramme de collaboration de la figure 11. Cette collaboration est intéressante pour plusieurs raisons. Premièrement, ce diagramme contient deux aspects, dont un, se référant au constructeur de la classe *Billing*, ce qui permet de couvrir tous les types de greffes. Deuxièmement, cet exemple contient quelques préconditions intéressantes permettant d'exécuter l'un ou l'autre des deux scénarios proposés. Ces conditions sont prises en compte par notre stratégie et doivent être testées adéquatement.

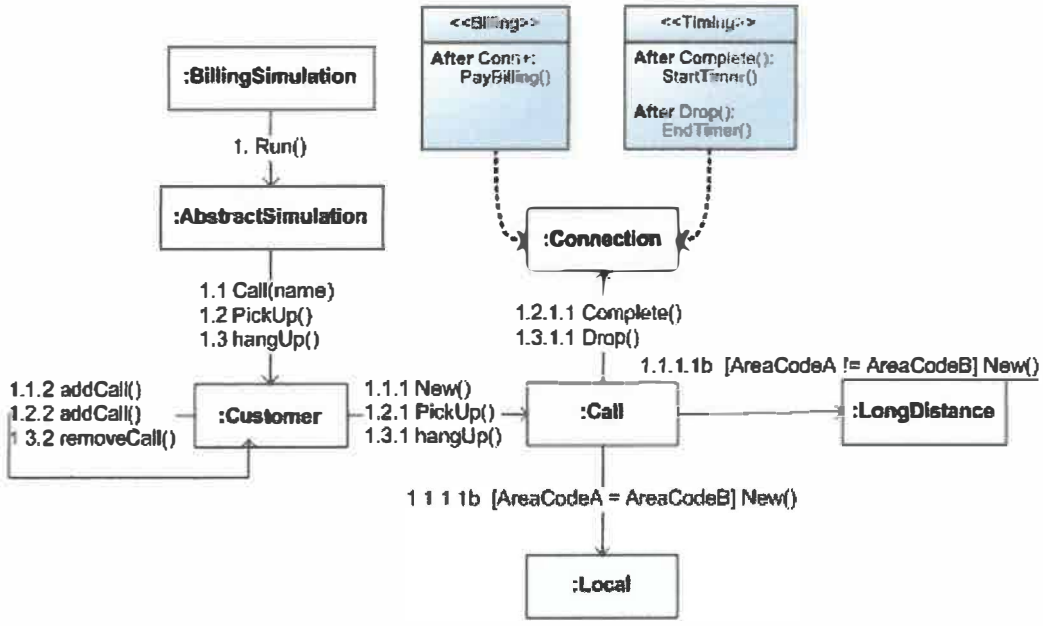


Figure 11 Diagramme de collaboration représentant la simulation d'appels téléphoniques.

Dans une première étape, nous commençons par créer l'arbre de messages du diagramme de collaboration à tester. En analysant l'arbre des messages, nous générons les séquences originales qui permettront de tester la collaboration entre classes afin de vérifier si l'implémentation des scénarios est conforme à la spécification. Ceci permettra par la suite, selon les critères introduits, de déterminer et de visualiser les scénarios modifiés par les aspects. L'intégration des aspects se fait selon un processus incrémental. Lorsque tous les aspects sont intégrés individuellement avec succès, nous re-testons le diagramme de collaboration dans son ensemble pour nous assurer que tout (aspects et classes collaborantes) fonctionne correctement. Cette étape nous permet également de déterminer les conflits éventuels qui peuvent être engendrés par les aspects.

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

1. Génération des graphes de contrôle des méthodes impliquées dans la collaboration.
2. Génération de l'arbre de messages de la collaboration.
3. Génération des séquences de test de base.

4. Test de la collaboration entre classes, basée sur les différents scénarios (critères *C1* et *C2*)
5. Intégration des aspects : Tant qu'il y a des aspects non intégrés
    a. Intégration d'un aspect (*en respectant la complexité*).
    b. Création des mini séquences selon le contrôle de l'aspect.
    c. Re-tester les séquences qui sont affectées par cette intégration.
    d. Si aucun problème, retour à l'étape 5.
6. Tester le diagramme de collaboration dans son ensemble incluant les aspects.
7. Fin

9.2 Graphe de contrôle

Les graphes de contrôle sont utilisés afin de construire une synthèse des algorithmes des opérations impliquées dans la collaboration. Ceci permet de présenter une vue globale du contrôle présent dans le diagramme. La figure 12 montre le graphe de contrôle en relation avec le diagramme de collaboration de la figure 11.

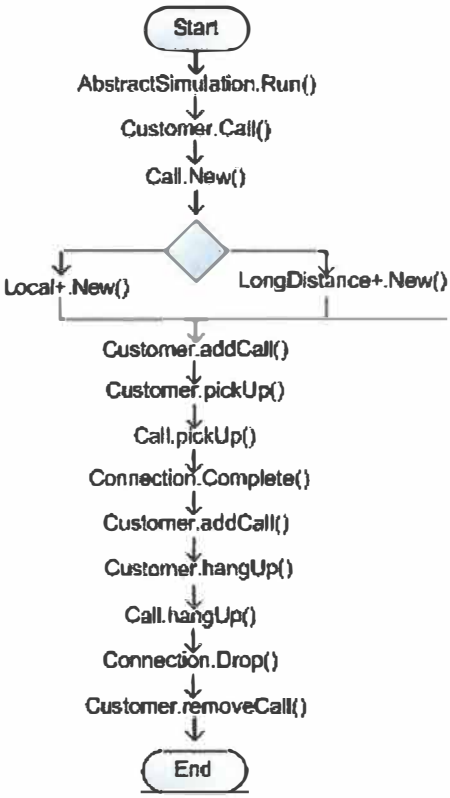


Figure 12 Graphe de contrôle pour le diagramme de collaboration *Telecom*.

9.3 Arbre de messages

Le graphe de contrôle de chaque opération impliquée dans la collaboration est traduit en une séquence principale (expression régulière). L'objectif à cette étape consiste à générer la séquence principale de chaque opération. À cet effet, nous utilisons la notation suivante : La notation {séquence}, exprime 0 ou plusieurs exécutions de la séquence. La notation (*séquence 1/séquence 2*) exprime une exclusion mutuelle lors de l'exécution entre la séquence 1 et la séquence 2. La notation [séquence] exprime que la séquence peut être exécutée ou non. Une fois les séquences des opérations créées, nous utilisons celles-ci comme base pour la construction de la séquence principale (correspondant à l'opération déclenchant la collaboration) et la génération de l'arbre de messages correspondant. Chaque message est remplacé par sa propre séquence. Le processus de substitution s'arrêtera au niveau des messages constituant les feuilles de l'arbre. À ce stade, nous ne tenons pas compte des aspects. Le tableau VI illustre la séquence principale de la collaboration représentée à la figure 11.

Tableau VI

Séquence principale pour le diagramme de collaboration *Telecom.*

AbstractSimulation.Run(), *Customer.Call(),Call.New(),* *(Local.New()/LongDistance.New()),* *Customer.addCall(), Customer.pickUp(),* *Call.pickUp(), Connection.Complete(),* *Customer.addCall(), Customer.hangUp(),* *Call.hangUp(),Conneciton.Drop(),* *Customer.removeCall()*

9.4 Séquences de test de base

La technique consiste à générer, à partir de l'arbre de messages, tous les chemins possibles partant de la racine jusqu'aux feuilles en tenant compte du contrôle. À chaque chemin correspondra une séquence de test particulière. Chaque séquence représente, en fait, un scénario particulier de la collaboration. Le tableau VII

présente chacun des messages impliqués dans le diagramme de collaboration ainsi que le numéro de nœud correspondant. Le tableau VIII illustre les séquences générées à partir du tableau VI. Nous procédons d'abord au test de ces séquences pour nous assurer que la collaboration fonctionne correctement.

Tableau VII

Messages de base du diagramme de collaboration avec leurs numéros de nœuds associés.

Node	Messages
1	*AbstractSimulation.Run()*
2	*Customer.Call()*
3	*Call.New()*
4	*Local.New()*
5	*LongDistance.New()*
6	*Customer.addCall()*
7	*Customer.pickUp()*
8	*Call.pickUp()*
9	*Connection.Complete()*
10	*Customer.hangUp()*
11	*Call.hangUp()*
12	*Conneciton.Drop()*
13	*Customer.removeCall()*

Tableau VIII

Séquences de base du diagramme de collaboration *Telecom.*

Scénario	Test	Séquences
1	#1	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 6 → 10 → 11→ 12 → 13
2	#2	1 → 2→ 3 → 5 → 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 13

9.5 Intégration des aspects

Lorsque toutes les séquences de base correspondant à la collaboration entre les classes sont générées, nous procédons à l'intégration des aspects. Les aspects sont intégrés de manière incrémentale, comme mentionné précédemment, dans le but de faciliter la détection d'erreurs. Nous avons adopté une stratégie itérative en

commençant avec l'aspect le plus complexe. Soutenues par les critères définis à la section 6, nous déterminons les séquences auxquelles se greffent des aspects. Ces séquences seront re-testées. Lorsque tous les aspects sont entièrement intégrés, nous testons toutes les séquences une fois de plus afin de vérifier que la collaboration, incluant tous les aspects, fonctionne correctement. Selon le diagramme de collaboration de la figure 11, l'aspect *Timing* introduit deux messages au niveau des séquences de base. Conséquemment, l'intégration des advice débutera avec cet aspect. L'ordre d'intégration des advice n'a pas d'importance dans ce cas-ci. Nous débutons avec l'advice *StartTimer* et terminerons avec *EndTimer*.

Tableau IX

Messages de l'aspect Timing ainsi que leurs numéros de nœuds.

Noeud	Messages
14	*Timing.StartTimer ()*
15	*Timing.EndTimer()*

Les séquences obtenues sont présentées dans les tableaux X et XI.

Tableau X

Séquences intégrant l'advice *StartTimer*

Scénario	Test	Nouvelles séquences
1	#3	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 14 → 6 → 10 → 11 → 12 → 13
2	#4	1 → 2 → 3 → 5→ 6 → 7 → 8 → 9 → 14 → 6 → 10 → 11 → 12 → 13

Tableau XI

Séquences intégrant l'advice *EndTimer*.

Scénario	Test	Nouvelles séquences
1	#5	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 15 → 13
2	#6	1 → 2 → 3 → 5→ 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 15 → 13

Sachant que tous les advice ont été intégrés individuellement, nous testons l'aspect dans son ensemble. Nous intégrons tous les messages aux séquences de base et nous les re-testons (tableau XII).

Tableau XII

Séquences intégrant l'ensemble des messages introduits par l'aspect *Timing*.

Scénario	Test	Nouvelles séquences
1	#7	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 14 → 6 → 10 → 11 → 12 → 15 → 13
2	#8	1 → 2 → 3 → 5→ 6 → 7 → 8 → 9 → 14 → 6 → 10 → 11 → 12 → 15 → 13

Par la suite, nous intégrons l'aspect *Billing*. Cet aspect est particulier au sens que son point de coupure affecte un constructeur de classe. Le problème principal vient du fait que cette classe est abstraite. Or, nous ne pouvons pas instancier ce type de classe. L'advice relié à ce point de coupure sera déclenché lorsqu'une des classes définissant la super classe sera instanciée. L'implémentation de l'interface de la classe *connexion* est faite par deux autres classes : *Local* et *LongDistance*. Ainsi, chaque instance de ces deux classes déclenchera l'exécution de l'advice défini dans l'aspect *Billing*. Les nouvelles séquences sont présentées dans le tableau XIV.

Tableau XIII

Message de l'aspect *Billing* ainsi
que de son numéro de nœud.

Noeud	Message
16	*Billing.PayBilling()*

Tableau XIV

Séquences intégrant l'advice *PayBilling*.

Scénario	Test	Nouvelles séquences
1	#9	1 → 2 → 3 → 4 → 16→ 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 13
2	#10	1 → 2 → 3 → 5 → 16→ 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 13

Lorsque l'intégration de tous les aspects est terminée, nous testons le système dans son entier en procédant au greffage de tous les aspects au diagramme de collaboration. Sachant que nous avons deux scénarios, deux derniers tests ont besoin d'être effectués (tableau XV).

Tableau XV

Séquences intégrant l'ensemble des messages introduits par l'aspect *Billing*.

Scénario	Test	Nouvelles séquences
1	#11	1 → 2 → 3 → 4 → 16 → 6 → 7 → 8 → 9 → 14 → 6 → 10 → 11 → 12 → 15 → 13
2	#12	1 → 2 → 3 → 5→ 16 → 6 → 7 → 8 → 9 → 14 → 6 → 10 → 11 → 12 → 15 → 13

Nous pouvons remarquer qu'une fois l'intégration finale effectuée, trois nouveaux messages ont été introduits dans les deux scénarios. Si nous calculons le pourcentage des nouveaux messages introduits sur l'ensemble des messages, nous obtenons :

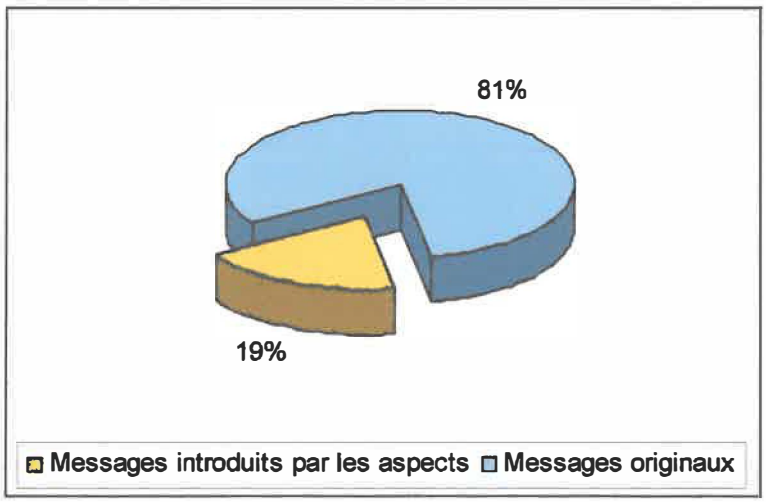


Figure 13 Rapport des messages aspects sur les messages originaux.

Nous constatons l'importance de tenir compte des aspects au niveau du test du diagramme de collaboration, car la figure 13 démontre que près de 20 % des messages proviennent de l'intégration des aspects.

CHAPITRE 10

VÉRIFICATION AUTOMATIQUE

Le processus de vérification permet essentiellement de vérifier si les séquences exécutées sont conformes à celles générées dans un premier temps, et si les résultats obtenus sont conformes à ceux attendus dans un second temps [Mass05-2]. Nous présentons, dans ce qui suit, les phases principales du processus de vérification. Cette section reprend l'exemple présenté à la figure 11. Pour chaque séquence générée S_i :

1. Instrumenter le programme sous test
2. Exécuter le programme sous test
3. Analyser les résultats obtenus

10.1 Instrumenter le programme sous test

Lorsque toutes les séquences ont été générées, nous pouvons amorcer le processus de vérification. Contrairement aux instrumentations traditionnelles, nous utilisons des aspects pour capturer dynamiquement la trace des messages exécutés. L'avantage de cette approche est qu'elle ne modifie en aucun cas le code source original de l'application que nous testons. L'instrumentation traditionnelle introduit de nouvelles instructions dans le programme à tester afin de capturer les messages qui sont exécutés. Elle permet, d'une façon générale, de récupérer des informations relatives à l'exécution du programme. Les fragments de code, introduits par instrumentation classique, ont la possibilité d'introduire involontairement de nouvelles fautes [Beiz90]. Dans la cadre de notre approche, nous générons un aspect pour chacune des séquences que nous voulons tester. Lorsque nous voulons tester une séquence précise, nous compilons le logiciel en intégrant l'aspect correspondant. Les aspects utilisés pour l'instrumentation sont automatiquement générés par notre outil et sont fonctionnels pour tous les logiciels AspectJ [Ajws05]. En fait, notre stratégie est générale et serait facilement adaptable pour n'importe quelle

implémentation orientée aspect. Lorsqu'un message impliqué dans une séquence est déclenché, l'aspect chargé du suivi conservera toutes les informations concernant l'exécution. Ces informations seront utilisées par la suite dans le processus d'analyse des résultats.

10.2 Exécuter le programme sous test

Lorsque la phase d'instrumentation est complétée, nous pouvons exécuter le logiciel. Cette étape consiste principalement à faire fonctionner le programme et tester une séquence particulière. Il revient au testeur de fournir correctement les données nécessaires au bon fonctionnement de la séquence sélectionnée. Lorsque les données ont été correctement fournies, nous pouvons lancer le logiciel avec le scénario que nous désirons tester.

10.3 Analyser les résultats

Maintenant que la séquence a été exécutée avec succès, un analyseur compare cette séquence avec celle attendue. Notre stratégie consiste essentiellement à découvrir trois types de fautes. Pour ce faire, nous avons développé un modèle de faute comportant trois niveaux. Ce modèle permet de classifier les trois types de fautes les plus souvent rencontrées. Le testeur peut alors orienter ses efforts plus efficacement dans la résolution de la faute. Ce modèle de faute comporte trois niveaux distinctifs :

1. Fautes basées sur les spécifications.
2. Fautes basées sur les préconditions.
3. Fautes basées sur le code source (les exceptions en Java).

Les fautes basées sur les spécifications

Ce type d'erreur survient lorsque certaines parties de la spécification ne sont pas transposées correctement dans le développement de l'application. Par exemple, un message (une méthode de classe ou un advice d'un aspect) qui serait manquant dans

l'implémentation. Dans ce cas, la liste des messages retournés sera différente de celle attendue. Pour tester ce type de fautes, nous avons volontairement l'appel d'une méthode dans la classe *Call*. La ligne 60 dans la figure 14 a été insérée comme étant un commentaire, ce qui empêchera l'exécution de la méthode *Complete*.

```
57    public void pickup()
58    {
59      Connection connection = (Connection)connections.lastElement();
60      //Connection.complete();
61    }
```

Figure 14 Provocation d'une erreur de spécification.

Fautes basées sur les préconditions

Cette étape vérifie toutes les préconditions définies dans le diagramme de collaboration. Le diagramme de collaboration présenté à la figure 11 suggère que deux séquences ont la possibilité d'être exécutées. La précondition reliée au message *New* des classes *Local* et *Longdistance* détermine si le scénario sélectionné concerne un appel local ou longue distance. Pour vérifier ce type d'erreur, le testeur peut volontairement fournir un jeu de données erronées (figure 15). Par exemple, lorsque nous voulons tester le premier scénario (appel local) nous pouvons fournir des données qui feront en sorte d'exécuter le second scénario (appel interurbain). Le logiciel sous test simulera un appel interurbain pendant que notre analyseur attend l'exécution d'un appel local. La précondition *AreaCodeA* = *AreaCodeB* ne sera pas respectée et une erreur sera lancée.

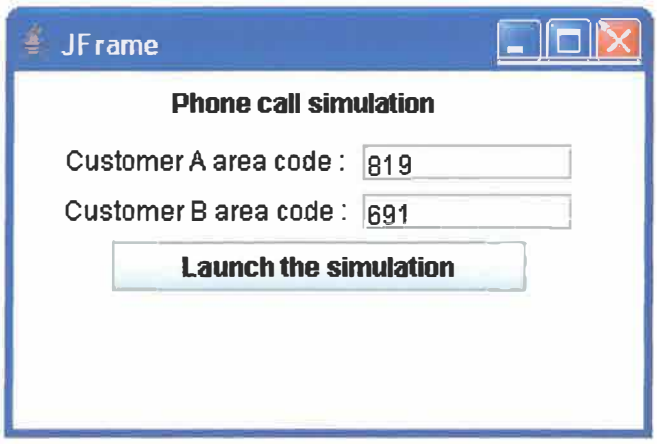


Figure 15 Insertion volontaire d'une erreur basée sur les préconditions (jeu de données erronées).

Fautes basées sur le code source (exceptions java)

Une des forces de java est sa capacité à gérer les erreurs lorsqu'elles surviennent. Subséquemment, il est possible avec AspectJ [Ajws05] d'intercepter ces erreurs. Lorsque nous créons automatiquement les aspects de test, nous générons aussi des points de coupure qui ont pour but de capter les exceptions lancées par java. Dans notre exemple (*Telecom*), nous avons inséré quelques lignes de code dans le but de provoquer intentionnellement une erreur d'exception (IOException dans l'aspect *Timing*.

```
45      private void StartTimer(Connection c)
46      {
47        getTimer(c).start();
48
49        /**
50       * Do an exception fault
51       */
52       BufferedReader in;
53       String f = "c:\\NonExistantFile.txt";
54       try
55        {
56          in = new BufferedReader(new FileReader(f));
57        } catch (FileNotFoundException e){
58          System.out.println("Can't open the file : " + f);
59          return;
60        }
61
62       /**
63       *End of fault code
64       */
65      }
```

Figure 16 Insertion volontaire d'une erreur basée sur les exceptions.

Le code introduit (lignes 49 à 64) de la figure 16, essaie d'ouvrir un fichier qui n'existe pas (*NonExistantFile.txt*). Puisque le fichier n'existe pas, Java lancera une faute du type I/O.

Toute déviation entre la séquence attendue et celle exécutée sera considérée comme une source de fautes. Dans un tel cas, la séquence ne sera pas considérée valide et ne sera pas incluse dans le calcul de la couverture de test. Pour être considérée valide, l'exécution d'une séquence doit être conforme aux critères de test établis au chapitre 7. Nous avons défini deux types de couverture de test. Le premier est en relation avec l'intégration simple des aspects tandis que le second porte sur l'intégration multi-aspects. Dans les deux cas, la couverture de test des séquences (SC) se calcule comme suit :

$$SC = \frac{Number\ of\ Executed\ Sequences\ (NES)}{Number\ of\ Generated\ Sequences\ (NGS)}$$

Lorsqu'aucune erreur n'est trouvée, la séquence sous test est marquée comme étant testée et le testeur est informé du pourcentage de test effectué.

CHAPITR 11

PRÉSENTATION DE L'OUTIL

Nous présentons dans ce chapitre l'outil développé pour supporter notre stratégie. Tous les éléments importants concernant sa conception ainsi qu'un exemple d'utilisation sont abordés. Nous présentons, dans un premier temps, une introduction à XML qui fut utilisé pour formaliser la structure des schémas (diagrammes de collaboration et aspect). Par la suite, nous illustrons sommairement, étape par étape, l'utilisation de notre outil. Nous générerons et testerons une série de séquences toujours en relation avec l'exemple présenté à la figure 11.

11.1 Introduction à XML

Nous présentons, dans ce chapitre une grammaire XML décrivant les aspects et les diagrammes de collaborations impliqués dans la génération des séquences de test. XML est un langage de description permettant de représenter schématiquement divers concepts. La syntaxe de XML est très stricte. XML ne fait rien à lui seul, car aucune balise n'est prédéfinie. C'est au développeur de les créer et de concevoir un logiciel permettant l'utilisation de sa structure XML. Il existe, cependant, sur les marchés divers correcteurs syntaxiques afin de vérifier la conformité d'un document XML. L'utilisation de XML, dans notre contexte, vise surtout à décrire formellement les objets (classes, aspects) impliqués dans la réalisation d'un scénario (base + extensions). Cette description concerne principalement le développement d'outils permettant de générer automatiquement les séquences de tests.

11.2 Structure XML des diagrammes de collaboration

L'étape de modélisation des diagrammes de collaboration consiste à inclure de manière formelle tous les éléments nécessaires à la génération des séquences. Les éléments importants relatifs à un diagramme de collaboration sont présentés dans le tableau XVI.

Tableau XVI

Structure XML des diagrammes de collaboration.

Balise	Description
<collaboration_diagram>	Balise racine du schéma.
<name>	Nom du diagramme de collaboration
<object_list>	Début de la liste des objets impliqués dans la collaboration.
<object>	Description individuelle d'une classe.
<object_name>	Nom de la classe.
<message_list >	Début de la liste des messages utilisés par la classe.
<message>	Description individuelle d'un message (méthode).
<message_name>	Nom d'un message.
<precondition>	Précondition (s'il y a lieu) d'un message.
<postcondition>	Post-condition (s'il y a lieu) d'un message.
<source>	D'où provient le message.
<destination>	Où se dirige le message.
<sequence>	Numéro de séquence du message.
<return_value>	Valeur de retour (s'il y a lieu) d'un message.
<parameter_list>	Début de la liste des paramètres utilisés par un message.
<parameter>	Description individuelle d'un paramètre.

La figure 17 présente un modèle générique d'une description XML pour un diagramme de collaboration. Tous les diagrammes de collaboration qu'utilise notre outil doivent être conformes à ce schéma.

```
<?xml version="1.0" encoding="utf-8" ?>
<!---->
<!--Début de la description d'un diagramme-->
<collaboration_diagram>
        <!--Nom du diagramme-->
        <name>NOM_DIAGRAMME</name>
        <!--Un D.C contient plusieurs objects, voici la liste-->
        <object_list>
                <!--Description d un object quelconque-->
                <object>
                        <!--Nom de l'objet (clase)-->
                        <object_name>NOM_OBJET</object_name>
                        <!--Un objet peut envoyer 1 à n messages, ceci est la liste-->
                        <message_list>
                                <!--Description d'un premier message-->
                                <message>
                                        <!--Nom du message-->
                                        <message_name>NOM_MESSAGE</message_name>
                                        <!--Pré-condition du message-->
                                        <precondition>PRECONDITION</precondition>
                                        <!--Poste-condition du message-->
                                        <postcondition>POSTCONDITION</postcondition>
                                        <!--D'où provient le message (objet)-->
                                        <source>SOURCE</source>
                                        <!--Où se dirige le message (objet)-->
                                        <destination>DESTINATION</destination>
                                        <!--Numéro de séquence du message-->
                                        <sequence>SEQUENCE</sequence>
                                        <!--Valeur de retour du message-->
                                        <return_value>VALEUR_RETOUR</return_value>
                                        <!--Un message peut prendre plusieurs paramètres, voici la liste-->
                                        <parameter_list>
                                                <!--Un premier paramètre-->
                                                <parameter>IDENTIFICATEUR</parameter>
                                        </parameter_list>
                                </message>
                        </message_list>
                </object>
        </object_list>
</collaboration_diagram>
```

Figure 17 Description XML (générique) d'un diagramme de collaboration.

11.3 Description XML des aspects

La modélisation des aspects fut une tâche plus ardue. Peu de travaux sur la modélisation des aspects sont disponibles. Nous avons donc étudié les mécanismes d'intégration des aspects afin de concevoir un modèle cohérent. Chaque aspect est représenté dans un fichier XML unique. La description des balises utilisées est présentée dans le tableau XVII.

Tableau XVII

Structure XML des aspects.

Balise	Description
<aspect>	Balise racine du schéma.
<aspect_name>	Nom de l'aspect.
<advice_list>	Début de la liste des advice inclus dans un aspect.
<advice >	Description individuelle d'un advice.
<advice_type >	Type de l'advice (BEFORE, AROUND, AFTER).
<advice_method>	Nom de la méthode de l'aspect.
<class_list >	Début de la liste des classes affectées par l'aspect.
<class>	Description individuelle d'une classe.
<class_name>	Nom de la classe.
<method_list>	Début de la liste des méthodes affectées par un advice.
<method_name>	Nom d'une méthode affectée par l'advice.

La figure 18 propose un exemple générique avec certains commentaires relatifs aux différents champs de la structure XML d'un aspect.

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Début de la description d'un aspect-->
<aspect>
  <!--Nom de l'aspect en question-->
  <aspect_name>NOM_ASPECT</aspect_name>
  <!--Un aspect peut contenir pusieurs adivce, en voici la liste-->
  <advice_liste>
    <!--La description d'un premier advice-->
    <advice>
      <!--Un advice peut être de trois types définit par l'OA-->
      <advice_type>"BEFORE", "AFTER", "AROUND"</advice_type>
      <!--Le nom de l'advice (nom de la méthode de l'asepct)-->
      <advice_method>METHODE_ADVICE</advice_method>
      <!--Un advice peut se greffer à plusieurs classes, en voici la liste-->
      <class_list>
        <!--La description d'une première classe-->
        <class>
          <!--Nom de la classe-->
          <class_name>NOM_CLASSE</class_name>
          <!--Un advice peut se greffer à plusieurs méthodes d'une seule classe-->
          <method_list>
            <!--Nom d'une première méthode-->
            <method_name>NOM_METHODE</method_name>
          </method_list>
        </class>
      </class_list>
    </advice>
  </advice_liste>
</aspect>
```

Figure 18 Description XML (générique) d'un aspect.

11.4 L'outil

L'outil développé est composé de plusieurs modules. Son architecture générale est présentée à la figure 19.

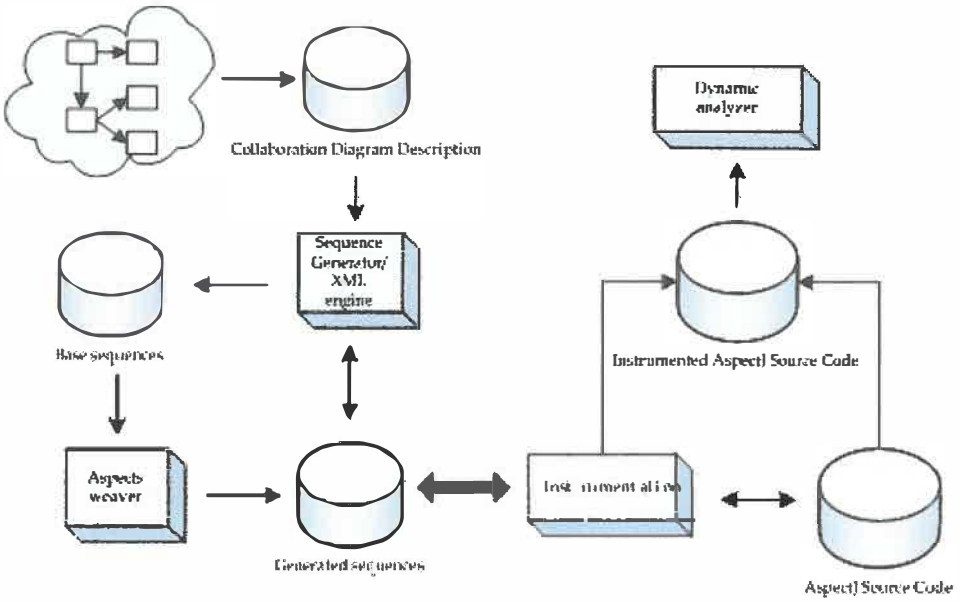


Figure 19 Architecture générale de l'outil.

Nous présentons, dans ce qui suit, un résumé des principales fonctionnalités de l'outil développé. Nous utilisons toujours la collaboration décrite à la figure 11. La démarche consiste en six étapes distinctes. Ces étapes débutent avec l'ouverture d'un diagramme de collaboration pour finir avec l'étape d'analyse des résultats. La figure 20 présente l'interface principale de notre outil. Toutes les fonctionnalités sont disponibles via la barre de menu en haut de l'écran. Nous utiliserons cette saisie d'écran pour guider l'utilisateur tout au long de l'expérimentation.

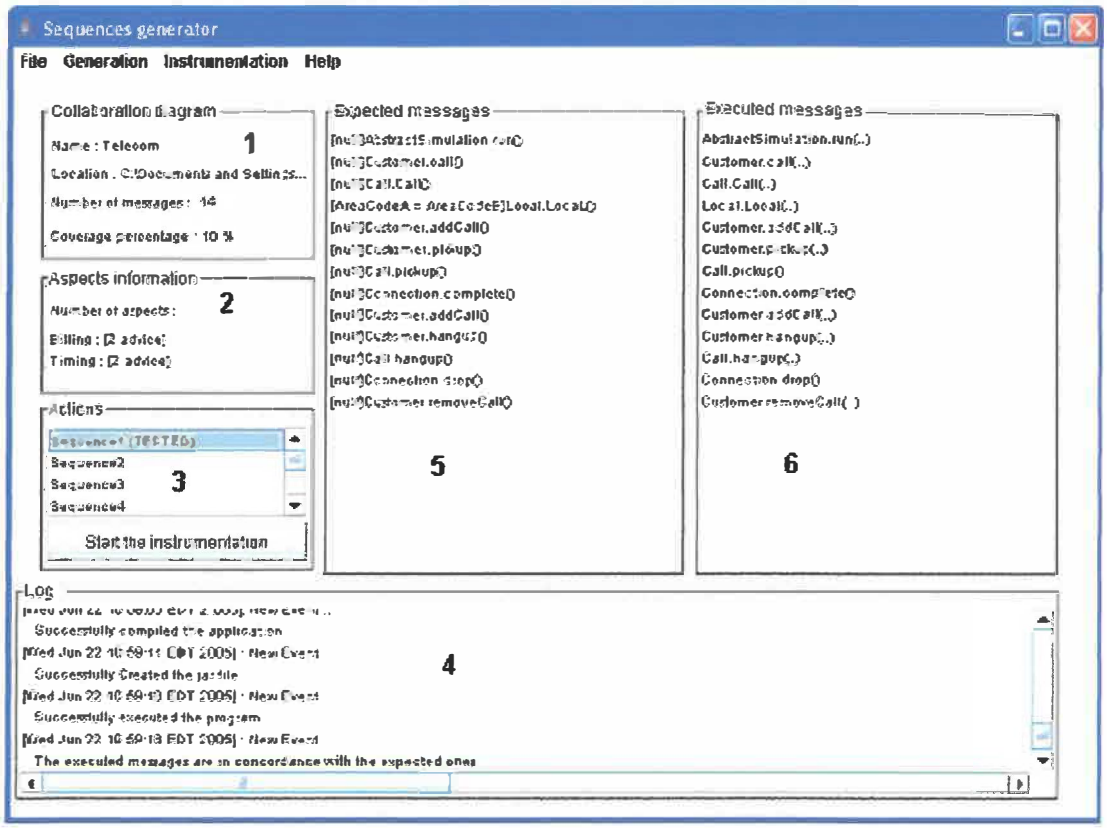


Figure 20 Interface principale de l'outil.

Nous donnons dans ce qui suit une description des étapes nécessaires couvrant les deux parties de notre stratégie de test.

1. Ouvrir le diagramme de collaboration et les aspects.
2. Générer les séquences de base.
3. Intégrer les aspects et générer les nouvelles séquences.
4. Instrumenter le programme sous test.
5. Exécuter le programme sous test.
6. Analyser les résultats.

11.4.1 Ouverture du diagramme et des aspects

Cette première étape vise à sélectionner les fichiers que l'outil utilisera pour générer et tester les séquences. Nous devons ici ouvrir le diagramme de collaboration à tester

ainsi que les fichiers où sont contenus les aspects qui devront être intégrés à la collaboration.

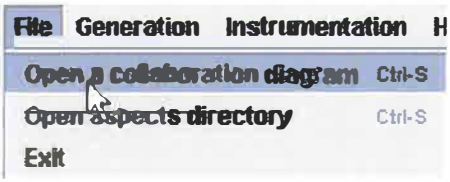


Figure 21 Ouverture d'un diagramme de collaboration.

Dans le menu *File* nous sélectionnons *Open a Collaboration Diagram* pour ouvrir le diagramme et *Open aspects directory* pour choisir l'emplacement des aspects. Dans notre exemple, nous ouvrons le diagramme de collaboration *Telecom* ainsi que les deux aspects *Billing* et *Timing*. Une fois les fichiers sélectionnés, certaines informations vont s'afficher dans la section (1) de l'interface principale. Le nom du diagramme de collaboration ainsi que le nombre de messages sont affichés. De plus, la section (2) présente les aspects ouverts ainsi que le nombre d'advice introduits par chacun d'eux. Un message est envoyé à la section (4) de l'interface pour informer que le diagramme et les aspects ont été ouverts correctement. Aucune opération n'a été effectuée à ce stade.

11.4.2 Génération des séquences de base

La génération des séquences de base se fait selon le processus présenté au chapitre 7. Pour réaliser ces étapes, nous allons dans le menu *Generation* disponible dans le haut de l'interface principale (figure 22). Cette étape ne tient pas compte des aspects. Un nouveau message est affiché à la section (4) de l'interface informant que les séquences de base ont été correctement générées.

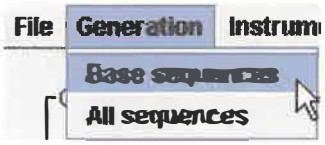


Figure 22 Génération des séquences de base.

11.4.3 Intégration des aspects

La troisième étape consiste à intégrer les aspects au diagramme de collaboration. Ces aspects sont analysés par notre moteur XML et greffés aux endroits appropriés aux séquences de base. Les nouvelles séquences sont ajoutées à la liste contenant les séquences de base. Pour effectuer cette opération, nous allons dans le menu *Generation* et sélectionnons *All Sequences.* À ce stade les séquences générées s'affichent dans la section (3) de l'interface principale. Concernant notre exemple *Telecom*, nous constatons que deux aspects ont été intégrés aux séquences de base pour générer un total de 10 séquences.

11.4.4 Instrumentation du programme sous test

Lorsque toutes les séquences sont générées avec les aspects, nous pouvons passer à l'étape de l'instrumentation de l'application sous test. Contrairement à l'instrumentation traditionnelle, nous utilisons des aspects afin de capturer l'exécution des messages. Un avantage important à propos de cette stratégie est que nous ne modifions en aucun point le code source de l'application que nous désirons tester. L'instrumentation traditionnelle introduit certains fragments de code dans le logiciel sous test. Ces fragments de code ont le potentiel d'introduire des erreurs [Beiz90]. Nous créons un aspect pour chacune des séquences à tester. Lorsque nous désirons tester une séquence particulière, nous compilons le logiciel en intégrant l'aspect correspondant. Lorsqu'un message impliqué dans une séquence est déclenché, l'aspect de test gardera une trace de cette exécution qui sera utilisée lors du processus d'analyse des résultats. Pour lancer la création automatique des aspects qui serviront à capturer les messages exécutés, nous allons dans le menu *Instrumentation* et sélectionnons *Creat traking aspects*... (figure 23). Cette opération aura pour but de générer un aspect par séquence. Ces aspects sont entreposés dans un dossier temporaire pour être utilisés à la prochaine étape. Une trace de cette opération est affichée dans le log de l'interface principale (section (4)).

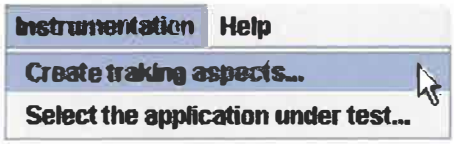


Figure 23 Instrumentation du logiciel sous test.

Maintenant, nous devons sélectionner le programme cible que nous devons tester. En Java, tous les programmes débutent leur exécution avec la fonction *Main* qui est contenue dans un fichier.*Java*. Ce que nous devons faire avant de commencer la procédure de test proprement dite est de choisir la classe d'entrée du programme. Pour ce faire, nous allons dans le menu *Instrumentation->Select the application under test...* (figure 23). Une copie du logiciel source est effectuée pour s'assurer que nous ne modifions pas le code source original.

11.4.5 Exécuter le programme sous test

Lorsque l'instrumentation a été correctement réalisée, nous pouvons exécuter le logiciel. Cela consiste principalement à exécuter le programme sous test et à tester une séquence précise. Pour ce faire, nous sélectionnons la séquence désirée dans la fenêtre gauche de l'interface principale (section (3)). Lorsque la séquence est sélectionnée, les messages impliqués dans la réalisation de cette séquence apparaissent dans la fenêtre *Expected messages* (section (5)). Les préconditions attachées aux messages sont également visibles au testeur. La figure 19 illustre l'interface principale où la séquence 1 a été sélectionnée pour être testée. Lorsque la séquence est choisie, nous lançons l'exécution du programme qui inclut maintenant l'aspect *Sequence1.aj* qui captera l'exécution des messages impliqués dans la séquence. Il revient au testeur de fournir les jeux de test appropriés qui assureront l'exécution du scénario correspondant à la séquence sélectionnée. Lorsque les entrées ont été fournies nous lançons l'exécution du logiciel en cliquant sur *Launch the instrumented software* dans la section (3). La fenêtre *Executed messages* affiche tous les messages exécutés lorsque l'exécution du logiciel se termine. Le mot « *TESTED* » s'inscrit au niveau de la séquence qui vient d'être testée (si celle-ci ne comporte aucune erreur). Après chaque test concluant, nous calculons le pourcentage

de couverture du test. Ce pourcentage informe le testeur de la portion qui est testée ainsi que de celle qui reste à vérifier. Par exemple, la figure 19 nous informe que 10 % des séquences ont été testés. Il est a noté qu'il peut être difficile d'atteindre une couverture de 100 % dans certains cas. C'est au testeur de choisir le pourcentage représentatif (niveau de confiance) lorsqu'il construit son plan de test. Ainsi, il se peut qu'un pourcentage inférieur à cent pour cent soit suffisant dans certains cas.

11.4.6 Analyse des résultats

Le processus d'analyse des résultats se fait selon le modèle de fautes présenté au chapitre 10. Les résultats de l'analyse sont présentés au testeur immédiatement après l'exécution de la séquence. L'information est affichée dans la fenêtre (4) de notre outil. Lorsqu'une séquence a été exécutée, quatre scénarios sont possibles.

Scénario 1 : *Aucune faute détectée*

C'est le scénario idéal puisque la séquence exécutée correspond à celle attendue. Dans ce cas, le message *The executed sequence is in accordance with the expected one* (voir figure 19) est affiché.

Scénario 2 : *Faute basée sur les spécifications*

Le logiciel découvre une erreur en relation avec les spécifications. La liste des messages retournés peut être identique ou différente. La figure 23 informe qu'une erreur de spécification a été trouvée. Cette erreur est le résultat de la faute introduite volontairement à la section 10 (méthode *Complete* de la classe *Connection*).

```
[Mon Jun 13 13:28:36 EDT 2005] : New Event...
  Successfully executed the program
[Mon Jun 13 13:28:36 EDT 2005] : New Event...
  The executed message do not correspond with the expected one [[null]Connection.complete()]
```

Figure 24 Message d'erreur concernant une faute basée sur une spécification.

Scénario 3 : *Faute basée sur une précondition*

Lorsqu'une précondition n'est pas respectée lors de l'exécution de la séquence, un message informe le testeur du problème. De plus, l'outil permet de repérer précisément où se trouve cette erreur. Selon la faute introduite au chapitre, le message d'erreur suivant (figure 24) informe le testeur que la précondition attendue n'est pas valide.

```
[Mon Jun 13 14:07:48 EDT 2005] : New Event...
  Successfully executed the program
[Mon Jun 13 14:07:48 EDT 2005] : New Event...
  Error in the precondition for the message : [AreaCodeA = AreaCodeB]Local.LocalL()
```

Figure 25 Message d'erreur concernant une faute basée sur une précondition.

Scénario 4 : *Faute basée sur une exception Java*

Le dernier scénario possible se produit lorsqu'une exception est lancée dans la séquence exécutée. Ainsi, lorsque nous avons provoqué volontairement une erreur (figure 25) dans la méthode *SetTimer* un message nous informe de l'origine de la faute ainsi que de son type.

```
[Mon Jun 13 14:48:55 EDT 2005] : New Event...
  Successfully executed the program
[Mon Jun 13 14:48:55 EDT 2005] : New Event...
  Exception found in the message : [null]Timing.StartTimer()
```

Figure 26 Message d'erreur concernant une faute basée sur une exception Java.

Nous avons illustré, grâce à cet exemple, le fait que notre stratégie, ainsi que l'outil développé, permettait de découvrir les erreurs relatives au modèle de fautes présenté à la section 10. L'outil permet de générer des séquences de test basées sur un diagramme de collaboration et d'y intégrer des aspects venant s'y greffer. Par la suite, nous pouvons tester ces séquences et vérifier l'exactitude des messages exécutés. Nous remarquons que la détection des erreurs se fait aisément et que le

testeur est automatiquement informé de l'origine et du type de l'erreur détectée. Ainsi, la correction de l'erreur se fait relativement facilement. La diminution des ressources nécessaires à la correction d'erreur est un résultat direct de cette démarche.

CHAPITRE 12

CONCLUSION

La programmation orientée aspect apporte de nouvelles abstractions au génie logiciel qui génère de nouvelles dimensions en termes de complexité. Les aspects introduisent de nouvelles dépendances. Les techniques de test orienté objet actuelles ne couvrent pas les caractéristiques du paradigme aspect. Les différents travaux effectués dans le domaine du test orienté aspect se sont intéressés soit au code source des aspects ou bien au code objet des applications sous test. Par ailleurs, d'autres travaux se sont intéressés à la génération de séquences de test à partir des diagrammes d'états-transitions. Ces travaux ont porté sur le comportement de classes auxquelles se greffe un aspect. Nous nous sommes intéressés, dans le cadre de ce projet, au comportement d'un groupe d'objets collaborant auxquels se greffent un ou plusieurs aspects. Notre démarche se base sur les spécifications décrites dans les diagrammes de collaboration UML. Elle permet de préparer le processus de test tôt dans le processus de développement.

La stratégie présentée se divise en deux grandes phases. La première consiste à générer les séquences de test. Les séquences sont générées, dans un premier temps, conformément aux différents scénarios de la collaboration sans les aspects. Cette étape permet de vérifier que la collaboration s'effectue correctement pour un groupe d'objets donnés. Par la suite, nous intégrons les aspects de manière incrémentale au diagramme de collaboration. Cette intégration s'effectue selon les critères présentés au chapitre 7. De nouvelles séquences sont créées et intègrent les messages introduits par les aspects. Les nouvelles séquences obtenues tiennent compte de la collaboration entre les objets, du contrôle lié à cette collaboration ainsi que des aspects qui s'y greffent.

La deuxième étape de la stratégie consiste en la vérification des séquences générées. Cette phase est supportée par une instrumentation du code AspectJ de l'application sous test. Ceci permet de vérifier que l'implémentation correspond aux spécifications de départ. Contrairement à l'instrumentation traditionnelle, nous utilisons des aspects pour capter les divers éléments utiles à notre analyse. L'analyse se fait en fonction d'un modèle de faute que nous avons développé pour les systèmes orientés aspect. Ce modèle comporte trois niveaux distincts. Le premier consiste à tester et vérifier les spécifications du diagramme de collaboration. Cette vérification vise à s'assurer que tous les détails du diagramme de collaboration ont été transposés en code source correctement. Le deuxième niveau s'intéresse aux préconditions décrites dans le diagramme. Lorsque nous exécutons le programme sous test, nous vérifions que toutes les préconditions relatives à un scénario donné sont respectées (c'est-à-dire vraies). Le dernier niveau de test vise à détecter les erreurs relatives au code source. Java permet une gestion des erreurs très développée. De plus, certaines constructions AspectJ permettent de capturer ces erreurs. Ainsi, lorsqu'une exception est lancée par un des messages impliqués dans la collaboration, nous pouvons la capturer et informer le testeur précisément où elle se trouve.

Le développement aspect étant encore à ses débuts, il fut difficile de trouver une application d'envergure conçue en utilisant ce paradigme. Cependant, il a été possible de mettre en œuvre notre stratégie de test sur un cas réel. Le cas choisi provient du site web AspectJ [Ajws05] et propose une simulation d'un système d'appels. Cette simulation permet de faire des appels locaux ou de longue distance. Les dispositifs relatifs au calcul du temps passé ainsi que de la facturation sont faits à l'aide de deux aspects. Nous avons mis à l'épreuve notre modèle de faute sur ce programme. Nous avons inséré volontairement des fautes dans le but de vérifier si notre outil était en mesure de capturer et d'informer le testeur correctement. Les résultats de cette étude ont permis d'observer des résultats très concluants. Toutes les erreurs introduites ont été interceptées par notre outil. À chaque fois, il était possible de localiser avec précision l'origine de l'erreur ainsi que son type en relation avec le modèle de fautes proposé.

Par ailleurs, nous envisageons d'adapter notre outil pour en faire un « plugin » qui pourrait s'intégrer directement dans la plateforme Eclipse. Ceci faciliterait grandement l'intégration de notre stratégie de test au sein d'un environnement de développement de programmes AspectJ.

BIBLIOGRAPHIE

[Abdu00] A. Abdurazik and J. Offutt, Using UML Collaboration Diagrams for Static Checking and Test Generation, In Third Internationl Conference on The Unified Modeling Language (UML '00), York, UK, October 2000.

[Alex04] R. Alexander, J. Bieman and A. Andrews, Towards the Systematic Testing of Aspect-Oriented Programs, Technical Report CS-4-105, Colorado State University, Fort Collins, Colorado, USA, March 2004.

[Aosd05] Aspect-Oriented Software Development Web Site (AOSD), http://.aosd.net/

[Ajpg02] T. AspectJ, The AspectJ™ Programming Guide, 2002.

[Ajws05] AspectJ Web Site, http://eclipse.org/aspectj/

[Badr05] M. Badri, L. Badri and M. Bourque-Fortin, Generating Unit Test Sequences for Aspect-Oriented Programs, 3rd International Conference on Information and Communication Technology (ICICT'2005), IEEE Computer Society Press, Cairo, Egypt, Décembre 2005.

[Badr04] M. Badri, L. Badri, and M. Naha, A use Case Driven Testing Process: Toward a Formal Approach Based on UML Collaboration Diagrams, Post-Proceedings of FATES (Formal Approaches to Testing of Software) 2003, in LNCS (Lecture Notes in Computer Science) 2931, Springer-Verlag, January 2004.

[Mass05-2]P. Massicotte, L. Badri and M. Badri, Aspects-Classes Integration Testing Strategy: An incremental approach. 2nd International Workshop on Rapid Integration of Software Engineering techniques (RISE 2005), in LNCS (Lectures Notes in Computer Science), Springer-Verlag, Heraklion, Crete, GREECE, September 2005.

[Mort04] M. Mortensen and R. Alexander, Adequate Testing of Aspect-Oriented Programs, Technical report CS 04-110, Colorado State University, Fort Collins, Colorado, USA, December 2004.

[Offu99-1] J. Offutt and A. Abdurazik, Generating Tests from UML Specications, In Second International Conference on the Unified Modeling Language (UML '99), Fort Collins, CO, October 1999.

[Offu96] J. Offutt and J. Voas, Subsumption of Condition Coverage Techniques by Mutation Testing, ISSE-TR-96-01, January 1996.

[Offu99-2] J. Offutt, Y. Xiong and S. Liu, Criteria for Generating Specification based Tests, In Engineering of Complex Computer Systems, ICECCS '99, Fifth IEEE International Conference, 1999.

[Sere03] D. Sereni and O. de Moor. Static analysis of aspects. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development. March 2003.

[Rrsc98] Rational Software Corporation. Rational Rose 98: Using Rational Rose, Rational Rose Corporation, Cupertino CA, 1998.

[Ubay02] N. Ubayashi and T. Tamai, Aspect-oriented programming with model checking, In Proceedings of the 1st international conference on Aspect-oriented software development, April 2002.

[Walk99] R. Walker, E. Baniassad and G. Murphy, An initial assessment of aspect-oriented programming, In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, May 1999.

[Wuye02] Ye Wu, Mei-Hwa Chen and Jeff Offutt, UML-based Integration Testing for Component-based Software, In Proceedings of the Second International Conference on COTS-Based Software Systems, September 2002.

[Xiet05] T. Xie, J. Zhao, D. Notkin, Automated Test Generation for AspectJ Programs, In Proceedings of the AOSD '05 Workshop on Testing Aspect-Oriented Programs (WTAOP 05), Chicago, March 2005.

[Xud05-1] D. Xu, Test Generation from Aspect-Oriented State Models. Technical Report, NDSU-CS-TR-05-XU02, Septembre 2005.

[Xud05-2] D. Xu and W. Xu, A Model-Based Approach to Test Generation for Aspect-Oriented Programs, AOSD'05 Workshop on Testing Aspect-Oriented Programs. Chicago, March 2005

[Xud04] D. Xu, W. Xu and K. Nygard, A State-Based Approach to Testing Aspect-Oriented Programs, Technical report, North Dakota University, Department of Computer Science, USA, 2004.

[Zhao02] J. Zhao, Tool support for unit testing of aspect-oriented software, In Proceedings OOPSLA' 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.

[Zhou04] Y. Zhou, D. Richardson, and H. Ziv, Towards a Practical Approach to test aspect-oriented software, In Proc. 2004 Workshop on Testing Component-based Systems (TECOS 2004), Net.ObjectiveDays, September 2004.