Ministère de l'Enseignement Supérieur et
de la Recherche Scientifique

UNIVERSITÉ 20AOUT 1955-SKIKDA جامعة 20 أوت 1955- سكيكدة

Département d'Informatique

MEMOIRE

Présenté en vue de l'obtention
du diplôme de Magister en Informatique
Ecole Doctorale en Informatique de l'Est (EDI Est)
Option : Génie Logiciel

THEME

Test Automatique Des Préoccupations Dans Les Langages A Aspects

Présenté par :
BOUTAGHANE Rafika

Devant le jury:

Président	Dr Nadir FARAH	MC A	Université Badji Mokhtar-Annaba
Rapporteur	Dr Djamel MESLATI	MC A	Université Badji Mokhtar-Annaba
Examinateur	Dr Mohamed REDJIMI	MC A	Université 20 aout 1955-Skikda
Examinateur	Dr Smain MAAZOUZI	MC A	Université 20 aout 1955-Skikda

Année 2010

RÉSUMÉ

La décomposition classique des programmes en objets présente l'avantage d'avoir un seul type d'entité ce qui a engendré des approches de test relativement claires. Il s'agissait de tester le comportement des objets et leurs interactions. Avec l'avènement de la séparation des préoccupations comme moyen privilégié de gestion de la complexité, les systèmes se retrouvent renforcés par un nouveau type d'entité matérialisant les préoccupations. Dans le cas de l'approche de programmation orientée aspects, les aspects ont pour rôle d'intercepter les événements générés par l'activité des objets et éventuellement d'intervenir pour altérer le comportement de ces objets en fonction des préoccupations que ces aspects incarnent. Cette façon d'opérer ouvre de nouvelles dimensions dans tout le cycle de vie du logiciel et en particulier l'activité de test.

Il existe actuellement des tentatives qui proposent des approches de test qui tiennent compte des préoccupations. Cependant, vu l'évolution constante des approches de séparation des préoccupations, les techniques de test ad hoc semblent encore immatures et beaucoup de travail reste à faire pour rattraper l'évolution du paradigme de séparation des aspects.

La problématique principale de ce mémoire est de faire le point sur les insuffisances des approches de test actuelles, de les comparer et de proposer une nouvelle technique convenable pour les programmes orientés-aspects.

Mots clés : Approche orientée-aspects, AspectJ, Séparation des préoccupations, Test logiciel

DÉDICACES

Je dédie ce mémoire

A mes chers parents, A mes sœurs, a mes frères,

A mon mari, a ma fille imane ,

A tous mes collègues de promotion ,

ET à tous ceux qui m'ont aidé de prés ou de loin à accomplir ce travail.

REMERCIEMENTS

Je remercie tous ceux qui ont participé de près ou de loin à l'achèvement de ce travail. J'exprime mes sincères remerciements à mon encadreur Dr Djamel Meslati pour m'avoir proposé le sujet de ce travail et pour avoir réorganisé et corrigé la version initiale de ce mémoire. Je remercie également les membres du jury, Dr Nadir Farah,, Dr Mohamed Redjimi et Dr Smain Maazouzi qui ont accepté d'évaluer ce travail.

TABLE DES MATIERES

TABLE DES ILLUSTRATIONS

LISTE DES TABLEAUX

INTRODUCTION

La présence de fautes introduites lors du développement d'un dispositif programmé est malheureusement courante. Elle doit être considérée avec beaucoup d'attention car notre vie de tous les jours dépend de plus en plus des systèmes logiciels et de lourdes pertes économiques, voire des issues fatales, peuvent être des conséquences de ces fautes. La validation des systèmes programmés avant leurs mises en service est essentielle et doit tenter d'atteindre un niveau de confiance satisfaisant. Ceci nécessite, de la part des opérateurs économiques, des académiciens et des chercheurs, le développement et l'amélioration de méthodes et d'outils adéquats pour la validation des systèmes logiciels.

Alors qu'il y a quelques décennies, un logiciel représentait quelques milliers de lignes de code, aujourd'hui ces produits comportent couramment plusieurs millions. Naturellement cette complexité entraîne une quantité plus importante d'erreurs lors du développement et en particulier lors de la programmation. Une étude menée par le « *National Institue of Standards and Technology* » évalue à 60 milliards de dollars par an le coût des bugs informatiques, dont 22 milliards de dollars pourrait être évités si l'industrie informatique améliorait ses équipements de test de logiciels.

Le test est une activité importante dans le processus de développement. Avec l'évolution des paradigmes, cette activité se retrouve face à de nouvelles dimensions qui soulèvent divers problèmes. C'est actuellement le cas pour le paradigme de séparation des préoccupations et en particulier la séparation des aspects. En effet, la puissance de ces paradigmes ramène aussi un lot de difficultés, en ce qui concerne le test, qui peut être considéré comme proportionnel à cette puissance.

Le but de la programmation orienté aspect (POA) est de répondre au problème des préoccupations qui s'entrecoupent (*crosscutting concern*) qui apparaissent dans les paradigmes de programmation actuels (procéduraux et orientés objets). Dans ces derniers, l'implémentation des besoins non fonctionnels est dispersée dans plusieurs modules (on parle d'*éparpillement* de code) et un même module renferme des éléments de plusieurs problématiques (on parle d'*enchevêtrement*). Par exemple, la sécurité et la synchronisation nous obligent à dupliquer des parties du code un peu partout dans différent modules fonctionnels d'un système. En conséquence, ce dernier ne bénéficiera pas d'une encapsulation adéquate ni au niveau des modèles de conception ni au niveau des langages de programmation. La POA ne tend pas à remplacer la programmation orientée objets mais bien à la compléter sur les éléments transversaux des logiciels.

Bien que ce paradigme procure des concepts puissants de modularité, il pose, cependant, de nouveaux problèmes, parmi lesquels la difficulté du test.

La décomposition classique des programmes en objets présente l'avantage d'avoir un seul type d'entité, ce qui a engendré des approches de test relativement claires. Il s'agissait de tester le comportement des objets et leurs interactions. Avec l'avènement de la séparation des préoccupations comme moyen privilégié de gestion de la complexité, les systèmes se retrouvent renforcés par un nouveau type d'entité matérialisant les préoccupations. Dans le cas de l'approche de programmation orientée aspects, les aspects ont pour rôle d'intercepter les événements générés par l'activité des objets et éventuellement d'intervenir pour altérer le comportement des objets en fonction des préoccupations que ces aspects incarnent. Cette façon d'opérer ouvre de nouvelles dimensions dans tout le cycle de vie du logiciel et en particulier l'activité de test. Cependant vu l'évolution constante des approches de séparation des préoccupations, les techniques de test ad hoc semblent encore immatures et beaucoup de travail reste à faire pour rattraper l'évolution du paradigme de séparation des aspects. Il existe actuellement des tentatives qui proposent des approches de test qui tiennent compte des préoccupations. Mais, la plupart sont inspirés des approches dédiées à la POO et souffrent d'insuffisances qui les rendent incapables d'être des références solides pour le test des programmes orientés aspects. La problématique principale de ce mémoire est de faire le point sur les insuffisances des approches actuelles et de proposer une nouvelle technique convenable pour les programmes orientés aspects.

Plus particulièrement, le travail s'intéresse aux points de jointures qui sont des points sensibles aux aspects dans le code de base. Une approche qui traite les cas de test liés à ces derniers et en particulier ceux qui sont dynamiques est nécessaire. Les points de jointures dynamiques provoquent des défauts liés à la structure du programme au moment de l'exécution. Pour cela, dans ce travail, nous nous intéressons au développement d'une technique de test structurelle qui tente de combler les insuffisances des approches de test actuelles. Concrètement, nous proposons un algorithme basé sur le principe de la séparation des préoccupations qui traite les trois points suivants :

1. Le tissage des tests d'aspects au graphe du flux de contrôle du code de base.
2. Évaluation complète des primitives dynamiques.
3. Définition de l'ordre de tissage des aspects.

Ce mémoire est structuré en 3 parties ; la première partie expose un état de l'art autour duquel se formule notre projet. Dans ce contexte, un tour d'horizon rapide des différentes méthodologies de test sera présenté. Ensuite, nous discuterons l'environnement dans lequel notre technique est appliquée pour combler les insuffisances des techniques existantes, à savoir le paradigme de programmation orientée aspect. Nous terminerons cette partie par

une synthèse des approches proposées actuellement pour le test des programmes orientés aspects.

La deuxième partie est l'étape la plus importantes de ce mémoire, elle consiste en une comparaison des approches de test orientés aspects existantes qui met l'accent sur les insuffisances de ces dernières. Suite à cela, nous présentons l'approche de test structurelle que nous proposons et qui est basée sur les points de jointures dynamiques. Leur fonctionnement est décrit dans le cadre d'un algorithme nommé GCC (graphe de contrôle complet).

La troisième partie joue le rôle d'un complément à la partie précédente. C'est la phase d'analyse et de conception. Nous y expliciterons, tous les détails de la modélisation ainsi que l'architecture du système. En fin la phase d'implémentation, qui met en œuvre notre système. Le mémoire se termine par une conclusion générale.

CHAPITRE 1
LE TEST LOGICIEL

Parmi les activités importantes et indispensables dans le cycle de vie d'un système logiciel on trouve les activités de test. Elles visent à s'assurer de la correction d'un système logiciel en ayant recours à plusieurs techniques de test allant jusqu'à la preuve du programme.

L'activité de test intervient à différents niveaux du développement allant des spécifications à la programmation puis l'intégration. L'objectif étant d'arriver à un produit à « zéro défaut », ce qui est une limite idéaliste vers laquelle on tend pour la qualité du logiciel. Actuellement, le budget consacré à l'effort de test et la maintenance atteint presque 80% du budget total alloué au projet.

Dans ce chapitre nous commencerons par la présentation du cycle de développement d'un logiciel. Puis nous exposons les principales définitions et caractéristiques du test.

1 Qu'est-ce qu'un logiciel ?

Un logiciel est un ensemble de programmes, qui permet à un ordinateur ou à un système informatique d'assurer une tâche ou une fonction [27, 30] (exemple : logiciel de gestion de la relation client, logiciel de production, logiciel de comptabilité logiciel de gestion des prêts, etc.).

Le terme *logiciel* est une traduction du terme anglais *Software*. Il constitue l'ensemble des programmes et des procédures nécessaires au fonctionnement d'un système informatique. Dans la famille des logiciels, on trouve par exemple des logiciels d'application qui sont spécifiques à la résolution des problèmes de l'utilisateur (progiciel, tableur, traitement de texte, grapheur, etc.), mais aussi des logiciels d'enseignement ou didacticiels, des logiciels de jeu ou ludiciel, etc. [41].

Le but du développement de logiciel est de produire un logiciel de qualité. De nombreux critères existent afin de définir la qualité d'un logiciel, nous nous y intéressons dans ce qui suit.

2 Qualité du logiciel

La qualité est ce qui rend une chose plus ou moins recommandable. C'est le degré plus ou moins élevé d'une échelle de valeurs pratique [42].

En génie logiciel divers travaux ont été menés sur la définition de la qualité du logiciel en termes de facteurs, qui dépendent, entre autres, du domaine de l'application et des outils utilisés. Les facteurs peuvent être classés en internes (visibles par les développeurs) et externes (visibles par les utilisateurs) [42].

Il existe plusieurs facteurs pour mesurer la qualité d'un logiciel parmi lesquels nous citons [42] :

- **Validité.** Aptitude d'un produit logiciel à remplir exactement les fonctions définies par le cahier des charges et les spécifications.
- **Fiabilité (ou robustesse).** Aptitude d'un produit logiciel à fonctionner dans des conditions anormales.
- **Extensibilité.** Facilité avec laquelle un logiciel se prête à une modification ou à une extension des fonctions qu'il assure.
- **Réutilisabilité.** Aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.
- **Compatibilité.** Facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.
- **Efficacité.** Utilisation optimales des ressources.
- **Portabilité.** Facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels.
- **Vérifiabilité.** Facilité de préparation des procédures de test.
- **Intégrité.** Aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés.
- **Facilité d'emploi.** Facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

Ces facteurs sont parfois contradictoires, le choix des compromis doit s'effectuer en fonction du contexte. Par exemple, la facilité d'emploi et la fiabilité peuvent être contradictoires. Dans une application du type *traitement de texte* c'est le premier facteur qui sera favorisé, alors que dans le cas d'un *pilotage d'usine*, c'est le deuxième facteur qui sera favorisé.

Pour développer un logiciel de qualité (qui satisfait les critères cité ci-dessus) il faut intervenir dans toutes les étapes du cycle de développement logiciel.

3 Le cycle de développement d'un logiciel

En peut résumer le cycle de développement d'un logiciel en sept étapes qui constituent le squelette du cycle logiciel (mais il est tout à fait possible de retrouver plus ou moins d'étapes selon le domaine d'application et les outils utilisés) [45].

3.1 Déterminer les besoins

La communication est primordiale afin d'arriver au résultat escompté par le client. Le client doit être bien identifié. Le client n'est pas nécessairement un connaisseur en informatique, c'est pourquoi la communication est importante ; les deux parties (clients et développeurs) doivent bien s'entendre sur ce qui doit être réalisé.

Cette étape doit spécifier les fonctions nécessaires pour répondre aux besoins du client. Les besoins existants doivent être traduits en besoins logiciels : les connaissances informatiques que doivent avoir les utilisateurs, les performances requises, les contraintes de réalisation, le nombre de fois que l'usager doit utiliser le système, les caractéristiques clés, obligatoires, etc.

Plusieurs moyens sont disponibles pour obtenir ces informations. Il est possible d'avoir un entretien avec les clients ou les utilisateurs. L'utilisation d'un questionnaire ainsi que l'observation du client dans ses tâches, peuvent aussi contribuer à répondre à ces questions.

Les besoins du client au niveau des interfaces, performances contraintes, qualités, ... doivent être traduits. Les deux parties doivent bien s'entendre afin de faciliter les étapes suivantes.

3.2 Conception préliminaire

Cette étape dite étape du « *quoi* » décrit de façon globale le produit. Les résultats de l'étape précédente sont utilisés afin de produire ce que le système doit faire. Cette étape doit permettre de savoir si le projet est faisable au niveau informatique et d'estimer le travail à faire. La solution doit être décomposée en une architecture modulaire.

3.3 Conception détaillée

La description du logiciel est faite dans cette étape. On doit spécifier davantage les détails du système afin d'arriver à une description très proche du programme final. C'est l'étape du «comment». Les modules de la section précédente doivent être décrits minutieusement. Les

fonctions doivent être décomposées en élément plus petit. Les données et algorithme doivent être décrits pour chaque élément. Les données d'entrée et de sortie doivent être précisées. Un plan de test doit être fait pour chaque module. Une bonne estimation du temps et du nombre de personnes nécessaires peut-être fait à la suite de cette étape.

3.4 Implémentation

Les modules de l'étape précédente doivent être codés (i.e. réalisés dans un langage de programmation). Souvent, c'est l'étape à laquelle les gens accordent relativement le plus d'importance malgré le fait que les autres étapes soient très importantes. Les autres étant souvent très négligés. Des tests de modules séparés sont effectués, il s'agit de test dits unitaires.

3.5 Intégration

Cette étape vérifie si l'architecture spécifiée dans la conception préliminaire est bien respectée. On rassemble les modules entre eux et on fait des tests progressivement en comparant les résultats obtenus à ceux attendus. Un travail de modification doit être fait si les résultats ne concordent pas avec ceux escomptés.

3.6 Test

Cette étape permet de savoir si on a bien produit le système que le client désirait et de savoir si on a répondu à ses besoins. Des tests de validation doivent être faits afin de vérifier si le logiciel est conforme aux spécifications préétablies. Le système doit être testé par d'autres personnes que ceux qui l'ont conçu et sur d'autres plateformes. Une formation des utilisateurs est faite à l'aide de documentation, des vidéos ou des cours.

3.7 Maintenance

Des anomalies peuvent être trouvées et une ou des personnes sont en charge de modifier les modules, documentations, ... afin de résoudre ces problèmes. Des nouvelles fonctionnalités peuvent être ajoutées. Des tests peuvent être refaits afin de s'assurer que les corrections n'apportent pas d'autre problème au système.

Chaque étape à un temps approximatif. Très souvent un temps important est alloué à l'implémentation. Même si aucun problème n'est rencontré, rien ne garantit, que le système est bien celui que le client désirait. Les temps approximatifs sont comme suit :

1. Déterminer les besoins, environ 15%
2. Conception préliminaire, environ 10%

3. Conception détaillée, environ 15%
4. Implémentation, environ 20%
5. Intégration, environ 10%
6. Test, environ 30%

4 Qu'est se que le test ?

Il existe plusieurs définitions pour le test logiciel, nous citons les suivantes qui donnent une idée complète sur la notion de test.

- Le test est un processus manuel ou automatique, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification [12].
- Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts [25].
- Le test est l'exécution ou l'évaluation d'un système ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond aux spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus [9].
- Le test est une technique de contrôle consistant à s'assurer, au moyen de son exécution que le comportement d'un programme est conforme à des données préétablies [26].
- Le test de logiciel est une activité de vérification. Son objectif est de détecter des fautes ou les inadéquations d'un logiciel. Il est nécessaire de préciser par rapport à quelles références on tente de dégager ces inadéquations les spécifications du logiciel, les normes ou règles portant sur son code ou les documents le concernant [6].

5 Terminologie lié au test

Pour cerner la notion de test, il convient de comprendre des notions connexes que nous présentons ci-après.

Spécification. En informatique, la spécification est un modèle d'un logiciel. C'est aussi l'étape en génie logiciel qui consiste à décrire ce que le logiciel doit faire. Plus généralement une spécification peut aussi représenter n'importe quel système dynamique comme un circuit électronique. On distingue.

- Les spécifications informelles, par exemple un texte en français
- Les spécifications semi-formelles avec une syntaxe plus précise ou comportant des diagrammes plus ou moins standardisés

- Les spécifications formelles qui présentent une syntaxe et une sémantique

Les spécifications font souvent partie d'une méthode de génie logiciel (comme Merise) ou d'une méthode formelle [31].

Satisfaction. Un programme satisfait sa spécification lorsqu'il est en tout point conforme aux exigences de celle-ci [31].

Vérification et Validation. Les activités de vérification et validation (V & V) s'appliquent à tous les processus, sous processus, activités ou tâches techniques intervenant dans les processus techniques d'ingénierie système [37].

La vérification a pour but de montrer que l'activité a été bien faite, en conformité à son plan de réalisation et qu'elle n'a pas introduit de défaut dans le résultat (ainsi définit-on progressivement bien le système). Elle peut se faire notamment sur les états intermédiaires successifs du produit de l'activité.

La validation a pour but de montrer que l'activité s'est conformée à son objectif, que le résultat de l'activité répond au besoin pour lequel l'activité a été faite (ainsi définit-on progressivement le bon système). Elle se fait en vérifiant la conformité du produit de l'activité à ses exigences de besoin (voir figure 1.1).

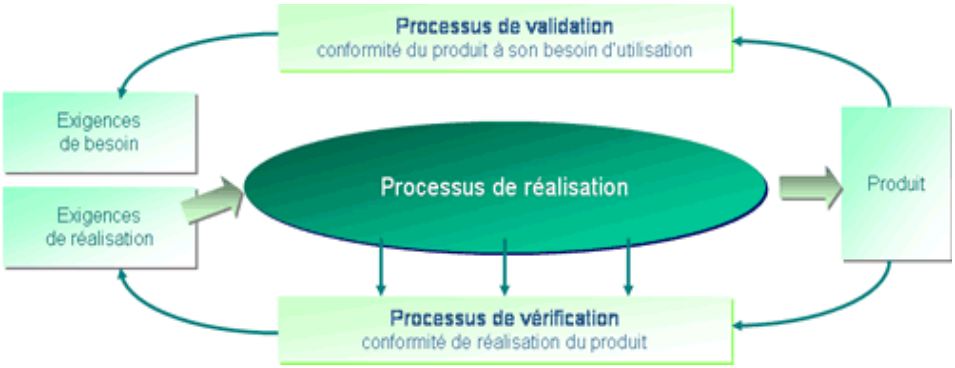


Figure 1.1 : Processus de vérification/validation

Faute / Défaillance / Erreur. En test logiciel, les termes de faute (ou défaut), défaillance et erreur sont différenciées mais liées par une relation de cause à effet.

L'*erreur* est commise par le développeur et désigne un état susceptible d'entraîner une *défaillance* du programme. Une *erreur* peut être décelée ou non décelée. Dans ce dernier cas, elle est dite *latente*. Le problème auquel nous sommes confrontés étant de les localiser et les corriger le plus tôt possible.

Quand l'erreur se manifeste dans le code, le terme associé est celui de *défaillance*. Une défaillance entraîne souvent l'apparition de fautes. Ainsi la relation entre ces trois concepts peut être qualifiée de cyclique comme le montre la figure 1.2 [28].

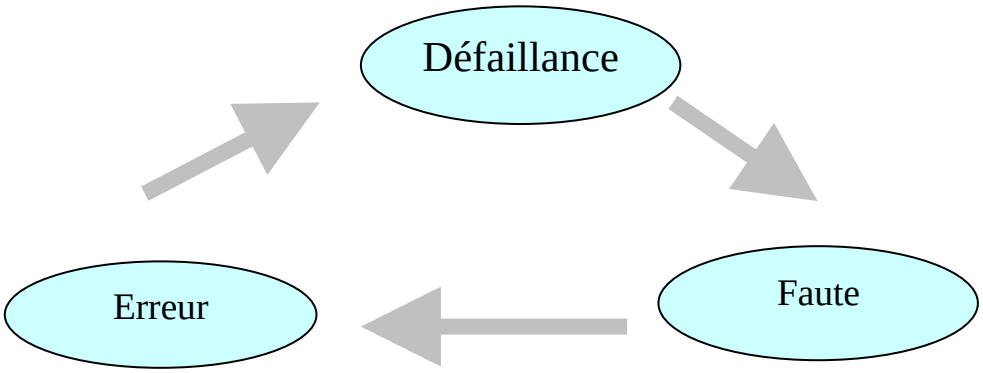


Figure 1.2 : Relation entre défaillance / faute / erreur.(manque de référence)

6 Caractéristiques et objectif du test

Le test est une recherche d'anomalie dans le comportement de logiciel. C'est une activité paradoxale il vaut mieux que ce ne soit pas la même personne qui développe et qui teste le logiciel. Un bon test est celui qui met à jour une erreur non encore rencontrée.

Le test n'a pas pour objectif de diagnostiquer la cause des erreurs, de corriger les fautes ou encore de prouver la correction d'un programme. Le test vise à mettre en évidence les erreurs d'un logiciel, il peut prouver la présence de l'erreur, mais il ne peut pas prouver leur absence. Le test vise à détecter les défauts avant qu'elles ne causent une panne du système produit et amener le logiciel testé à un niveau acceptable de qualité (après la correction des défauts identifiés). Le test doit avoir l'efficacité comme caractéristique et cela en respectant le temps et le budget alloués. Comme ultime objectif, le test doit conduire à la compilation d'un ensemble de situations d'erreurs devant servir à leur prévention (par des actions correctives et préventives).

Les phases de test dans le cycle de développement d'un produit logiciel permettent d'assurer un niveau défini de qualité en accord avec le client. Une procédure de test peut donc être plus ou moins fine, et par conséquent l'effort de test plus ou moins important et coûteux selon le niveau de qualité requis [32].

7 Le test dans le cycle de développement

Les activités liées au test se déroulent tout au long du cycle de vie et interagissent fortement avec les activités de développement. En distingue cinq étapes du processus de test [43].

a- Le test unitaire. On teste les composants individuels de manière à s'assurer qu'ils fonctionnent correctement. On traite chaque composant comme une entité isolée, sortie du contexte des autres composants.

b- Le test des modules. Un module est un ensemble de composants, comme un objet, ou une liste de procédures et fonctions. Un module encapsule des composants ayant trait les uns avec les autres. Il sera donc possible de le tester indépendamment des autres modules du système.

c- Le test des sous systèmes. Il s'agit de tester une série de modules ayant été intégrés de manière à constituer un sous système. Il est possible de concevoir et implémenter des sous systèmes indépendants les uns des autres. Les problèmes les plus fréquents, dans les grands systèmes logiciels, sont dus à des interfaces qui ne se correspondent pas. Le test d'un sous système devra donc se concentrer sur la détection d'erreur au niveau des interfaces.

d- Le test du système. On intègre tous les sous systèmes pour constituer le système lui-même. Le processus de test doit alors essayer de mettre en évidence des erreurs dues à des interactions imprévues entre sous systèmes. Il doit aussi valider le fait que le système correspond bien aux besoins fonctionnels et non fonctionnels exprimés en début de projet.

e- Le test d'acceptation. C'est la dernière étape avant l'acceptation du système et sa mise en œuvre opérationnelle. On teste le système dans les conditions définies par le futur utilisateur, plutôt que par le développeur. Les tests d'acceptation révèlent souvent des omissions ou des erreurs dans la définition des besoins. Il est possible que les besoins énoncés au début du projet ne reflètent pas les fonctionnalités réelles ou les vraies performances requises par l'utilisateur. C'est le genre de situation que met en évidence le test d'acceptation.

La figure 1.3 met en correspondance les tests avec les étapes du développement.

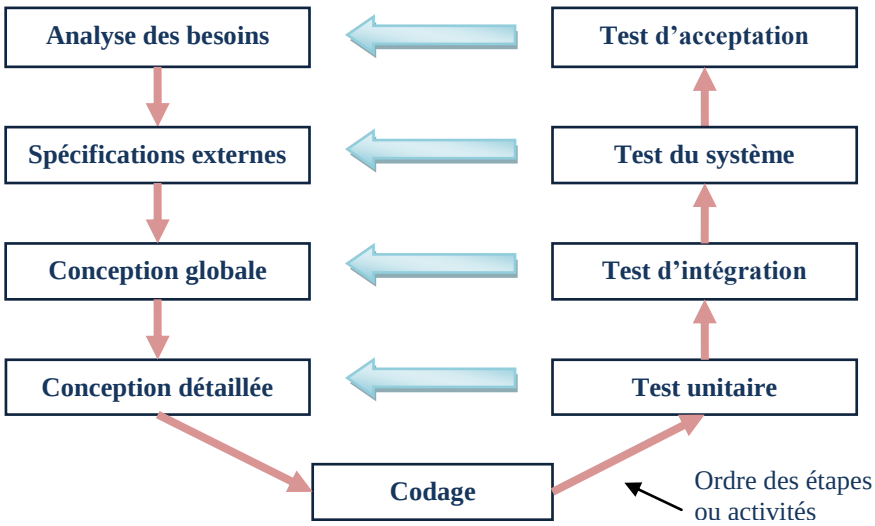


Figure 1.3 : Le test dans le cycle de développement logiciel

8 Classification des méthodes de test

Il existe différentes classes de méthodes de test selon que le programme est exécuté ou non pendant le test. Deux de ces méthodes sont clairement différenciées. Une troisième méthode existe mais son classement reste encore un sujet de désaccord entre les spécialistes du test. les tests statiques sont particulièrement utiles dans les cas de non déterminisme exécutoire comme lors de l'utilisation du parallélisme.

8.1 Les méthodes de test statiques

Les méthodes de test statiques consistent en l'analyse textuelle du code logiciel afin d'y détecter des erreurs, sans l'exécution du programme. L'idéal, lors de ce test, est de réunir quatre intervenants : un Modérateur, le Programmeur, le Concepteur et un Inspecteur. Parmi les techniques de test statique, on peut citer les suivantes :

1- Revue ou inspection. Examen détaillé d'une spécification, d'une conception ou d'une implémentation par une personne ou un groupe de personnes, afin de détecter des fautes, des violations de normes de développement ou d'autres problèmes [12].

2- Lectures croisées. Ce mode permet d'améliorer la qualité de lecture d'un document. Cela permet en outre de faire circuler l'information au sein du groupe de développements. Pour effectuer une lecture croisée d'un document, il faut associer un lecteur à chacun des auteurs. Après retours successifs l'un à l'autre, un certain nombre de modifications à effectuer sur le document sont mises en évidence, notées et commentées. A la fin du cycle « auteur/lecteur », le document est corrigé [28].

3. L'analyse d'anomalies. Un certain nombre d'anomalies peuvent être facilement détectées dans les logiciels via les analyses faites à la compilation. Ces anomalies peuvent être par exemple la violation de certaines règles raisonnables de typage, des utilisations impropres de pointeurs, la présence d'expressions ayant clairement une valeur constante, des défauts de portabilité, des sorties ou entrées de boucle hors des points standard, la déclaration d'arguments ou de librairies jamais utilisés, etc. [6]. Certaines de ces anomalies n'impliquent pas nécessairement une faute logique, mais elles peuvent en révéler avec une assez forte probabilité. On peut remarquer également que l'utilisation d'un langage de programmation fortement typé éliminera, de fait, un grand nombre de ces fautes [44].

Dans ce qui suit, nous décrivons trois approches d'analyse d'anomalies.

- **Analyse des anomalies par graphe de contrôle**. Dans un graphe de contrôle, les nœuds du graphe représentent les séquences indivisibles maximum ayant un seul point d'entrée. Les arcs matérialisent les transferts de contrôle et les étiquettes identifient des conditions de transfert. On commence par identifier les blocs d'instructions indivisibles maximaux. Ce sont les portions de code linéaires maximales que l'on exécute toujours du début à la fin. Ils sont constitués d'une suite d'instructions élémentaires où le contrôle passe successivement de l'une à la suivante sans déroutement (choix) possible. Chaque bloc n'a qu'un point d'entrée (son début) [44].
- **Analyse des anomalies par flot de données**. En annotant le graphe de contrôle, décrit précédemment, par des informations pertinentes quant à la manipulation des variables, on modélise le flot des données. On ne retient que les informations d'assignation (définition) et d'utilisation d'une variable affectation et accès à la valeur. Les anomalies détectables à partir de flot de données sont principalement [44] :
    - Utilisation d'une variable sans assignation préalable
    - Assignation d'une variable sans utilisation ultérieure
    - Redéfinition d'une variable sans l'avoir utilisé
- **Analyse des anomalies par système de transitions. Un système de** transition décrit des états qui sont des abstractions des états les plus généraux d'un processus et des transitions qui modélisent les actions (événements) provoquant les changement d'états du processus modélisé. Ces transitions peuvent être étiquetées, ce qui permet de les nommer ou de caractériser les transitions de

même nature. La représentation d'un système de transitions étiquetées est un graphe orienté (étiqueté) dont les nœuds (sommets) sont les états du processus modélisé et les arcs (étiquetés) sont les transitions (étiquetées) entre états. Ainsi un système de transitions dénote un automate fini.

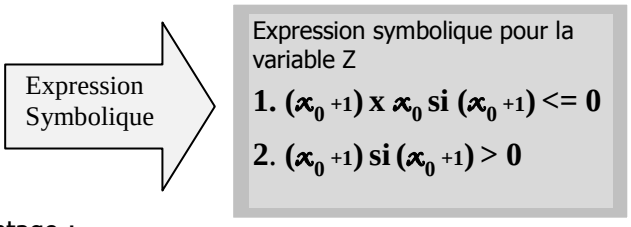
Les systèmes de transitions sont particulièrement adaptés à la modélisation de processus concurrents, notamment la modélisation des protocoles de communication. Les systèmes de transitions et leurs variantes sont, à l'heure actuelle, le formalisme le plus utilisé de modélisation de la concurrence [44].

4- L'évaluation symbolique. Elle consiste à simuler l'exécution du programme sur des données symboliques. Pour tout chemin sélectionné du graphe de contrôle, les données d'entrée sont représentées par des symboles. On obtient ainsi des expressions correspondant au texte du programme. L'évaluation symbolique produit, pour chaque variable de sortie du programme, une expression retraçant les calculs effectués pour l'obtenir (le long de ce chemin) et les conditions associées (sur le chemin) [6]. Il ne s'agit pas d'un test dynamique car l'évaluation symbolique ne requiert pas l'exécution du logiciel [44].

Exemple

Soit la partie du code suivant

```
read(x);
y  = x+1;
if y <= 0 then z  = y * x
else z  = y;
```



L'évaluation symbolique a comme avantage :

1. Méthode efficace et peu coûteuse.
2. La plus part des erreurs sont détectées lors de contrôle statiques (plus de 50%).

Elle présente, cependant, les inconvénients :

1. Elles ne permettent pas de valider le comportement d'un programme.
2. Lors d'une modification du programme, il est souvent difficile de réutiliser les tests précédemment validés avec la nouvelle version.
3. Mise en place lourde, tâche fastidieuse.

8.2 Les méthodes de test dynamiques

Les méthodes de test dynamiques consistent en l'exécution du programme à valider (sous test) à l'aide d'un jeu de test. Elles visent à détecter des erreurs en confrontant les résultats obtenus par l'exécution du programme à ceux attendus par la spécification de l'application. Le test dynamique suppose l'exécution des quatre étapes suivantes.

a- Sélection de jeux de tests. Cette étape consiste à trouver des données d'entrée, des scénarios pour exécuter le test ainsi que le résultat attendu.

b- Soumission du jeu de tests. Afin d'exécuter les tests retenus lors de l'étape précédente, un programme de test est exécuté avec les données de test. Ce programme peut être le logiciel qu'on veut tester dans le cas du test système ou du test d'acceptation. Pour le test unitaire et le test d'intégration, le programme de test est un pilote de test qui doit mettre en exécution la partie de logiciel qu'on souhaite tester.

c- Dépouillement des résultats. Il consiste à décider du succès ou échec du jeu de test. Il se pose alors le problème de l'oracle car la décision n'est pas toujours simple, ni même possible. Un *oracle* est une fonction qui doit permettre de distinguer si une sortie du logiciel est correcte.

Si l'oracle a détecté une défaillance, il faut la corriger et ré-exécuter le même cas de test pour vérifier que la correction a effectivement éliminé la faute.

Si l'oracle n'a pas détecté de défaillance, on a deux directions soit le critère d'arrêt est satisfait et la phase de test du logiciel est terminée ; soit un nouveau cas de test sera considéré.

d- évaluation de la qualité des tests effectués (critères d'arrêt). On peut rarement effectuer un test exhaustif. Dans la plupart des cas, on doit se contenter d'un test moins coûteux et il faut donc avoir un critère pour décider quand le test est satisfaisant. La figure 1.4 donne l'algorithme du test dynamique.

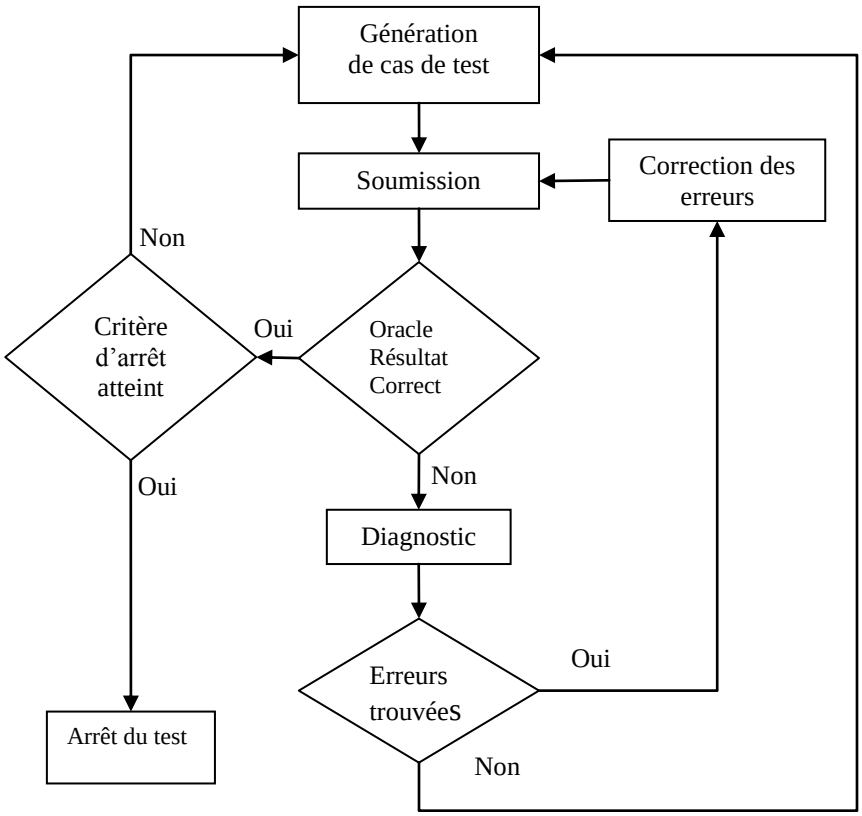


Figure 1.4 : Les étapes du test dynamique

8.2.1 Le test structurel (boite blanche)

Le test structurel est basé sur le détail du système. Il est effectué sur des produits dont la structure interne est accessible. Il est dérivé en examinant l'implémentation du système (langage source, conception de base de données, etc.).

Il s'intéresse principalement aux structures de contrôle et aux détails procéduraux. Il permet de vérifier si les aspects intérieurs de l'implémentation ne contiennent pas d'erreurs de logique. Il vérifie si toutes les instructions de l'implémentation sont exécutables.

Donc, le test structurel est caractérisé par une sélection des jeux de tests reposant sur la description de la structure du logiciel à tester. Deux types principaux de description sont utilisés le graphe de contrôle et le flot de données.

Ce test repose sur la sélection de données d'entrée qui déclenchent l'exécution de certains de ces chemins (Fig. 1.5). Il n'est pas nécessaire et généralement pas possible de parcourir tous les chemins. En pratique, on se donne des critères de couvertures plus restreints. En effet, la présence de boucles peut facilement conduire à un trop grand nombre de chemins, pouvant être infini, et l'existence de chemins impraticables empêche de couvrir tous les chemins.

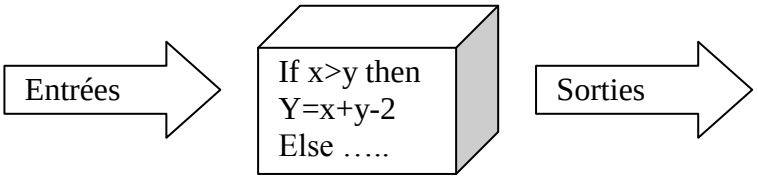


Figure 1.5 : Le Test structurel

Les critères de couverture de la structure peuvent se baser sur le flot de contrôle ou sur le flot de données.

Critères de couvertures basées sur le flot de contrôle. Un graphe de contrôle est un graphe orienté, composé d'un ensemble de nœuds reliés par un ensemble d'arcs orientés. Un arc (branche) décrit une séquence linéaire contiguë d'opérations (instructions) entre deux nœuds. Un nœud dénote un point d'entrée du programme (nœud d'entrée), un point de sortie (nœud de sortie) ou un transfert de contrôle (branchement, nœud interne).

Les critères de couvertures de graphe de contrôle les plus connus sont :

Couverture de toutes les instructions. Chaque bloc d'instructions doit être atteint par au moins un chemin parmi les chemins exécutés. La figure 1.6 donne une illustration des blocs de base dans un graphe de contrôle. Ce critère est assez faible par exemple, pour une portion de programme de la forme *bloc1 if expr then bloc2, bloc3* ; il n'est pas nécessaire d'exécuter un test rendant l'expression « expr » fausse.

Couverture de tous les enchaînements ou chaque arc doit être utilisé par au moins un chemin parmi les chemins exécutés. Cette fois, l'enchaînement correspondant au « else » implicite de l'exemple précèdent (« expr » fausse) doit être couvert. Cependant, rien n'impose de passer plus d'une fois dans une boucle.

Couverture des chemins avec boucle. On impose, ici, que tous les chemins possibles passent de 0 a i fois dans chaque boucle.

Couverture de tous les chemins. C'est la couverture la plus utilisée. Cependant, il est rare de pouvoir atteindre des niveaux de couvertures de chemins élevés du fait des considérations déjà citées. (Ce critère est rarement envisageable) [44].

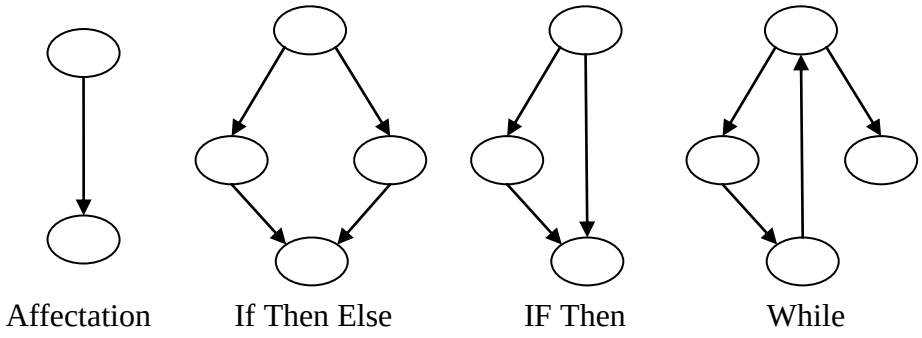


Figure 1.6 : Les éléments d'un graphe de contrôle

Critères de couvertures basées sur le flot de données. Plus utilisés, ils reposent sur la notion suivante : Une utilisation d'une variable est atteinte par une assignation (définition) de cette variable s'il existe un sous chemin du graphe de contrôle qui va de cette assignation à cette utilisation et qui ne comporte pas d'autre assignation de la variable. Dans ce cas les critères de couverture pourront être :

1. **Couverture de toutes les assignations.** Ici, chaque assignation d'une variable doit être exécutée au moins une fois par un chemin atteignant ensuite une utilisation de la variable.
2. **Couverture de toutes les utilisations.** Ici chaque assignation d'une variable doit être exécutée au moins une fois pour toutes les utilisations qu'elle atteint; de plus, tous les arcs issus de ces utilisations doivent être couverts.
3. **Couverture de toutes les P-Utilisations.** C'est une version affaiblie du critère précédent où on se limite aux utilisations atteintes qui apparaissent dans une condition.
4. **Couverture de tous les D-U-chemins.** C'est est une version renforcée du critère toutes les utilisations où de plus, chaque sous chemin entre une assignation et une utilisation qu'elle atteint doit être exécuté, en se limitant aux sous chemins passant 0 ou 1 fois dans chaque boucle [44].

Le test structurel s'appuie sur l'analyse du code source de l'application (ou d'un modèle de celui-ci), pour établir les tests en fonction de critères de couverture. On distingue trois méthodes de test structurelles :

a. Basés sur le graphe de flot de contrôle

b. Basés sur la couverture du flot de données

c. Basés sur les fautes (test par mutants)

Les méthodes de test structurel permettent d'avoir l'assurance que tous les aspects (en fonction des critères choisis) d'un logiciel ont été validés. En conséquence, elles sont indispensables et doivent faire partie de l'effort global de test. Néanmoins on remarquera que la sélection des tests dépend des graphes donc de la structure des textes des programmes. Chaque modification du logiciel devra donc amener une révision de ces tests. D'autre part, ces tests valident ce que fait le logiciel et non pas ce qu'il est censé faire. Il faudra donc également utiliser le test fonctionnel.

Comme avantage du test structurel nous avons :

- possibilité de fixer finement la valeur des entrées pour sensibiliser des chemins particuliers du code.
- critères de couvertures divers et précis.

L'inconvénient du test structurel réside dans l'absence de réutilisation. En effet la **c**onception des tests est faite pour un code et ne peut être réutilisée pour tester ce code après sa modification [7].

8.2.2 Le test fonctionnel (boite noire)

Ce type de test s'intéresse aux besoins fonctionnels du système. Il comporte deux étapes importantes :

1. L'identification des fonctions que le système est supposé offrir.
2. La création de données de test qui vont servir à vérifier si ces fonctions sont bien réalisées par le système.

Le processus du test fonctionnel s'effectue en trois activités :

1. Le testeur sélectionne d'abord des tests assurant un bon échantillonnage des comportements possibles.
2. Ensuite il exécute ces tests sur le programme et observe son comportement.
3. Finalement il compare les comportements observés avec les comportements attendus.

Le test en boite noire a comme buts de [38] :

1. Vérifier le comportement d'un logiciel par rapport à sa spécification (fonctions non-conformes ou manquantes, erreurs d'initialisation ou terminaison du logiciel).

2. Vérifier le respect des contraintes (performances, espaces mémoires, etc) et de facteurs de qualité associés au logiciel (portabilité, etc).

Le test fonctionnel peut s'appliquer méthodiquement à partir d'une spécification informelle (langage naturel), ou une spécification formelle (algébrique)

a. Cas d'une spécification informelle. Dans ce type de spécification il faut d'abord identifier les fonctionnalités requises ce qui impose au minimum de structurer la spécification d'une façon adéquate. Une méthode de test peut alors être associée à la spécification considérée.

b. Cas d'une spécification formelle. Les méthodes formelles sont des techniques permettant de raisonner rigoureusement sur les programmes informatiques, afin de démontrer leur correction. Elles se basent sur des raisonnements logiques et mathématiques. Elles peuvent être utilisées pour donner une spécification du système que l'on souhaite développer ou tester à divers niveaux de détails. Une spécification formelle du système est basée sur un langage formel dont la sémantique est bien définie. Un autre avantage de l'utilisation des spécifications formelles réside dans le fait que l'étape d'oracle et relativement plus simple à mettre en œuvre. En effet, un langage de spécification formelle exécutable a la possibilité de générer automatiquement les résultats attendus. La tâche du testeur se limitera donc à confronter ces derniers aux résultats obtenus après une session de test [46].

Le test fonctionnel vise à examiner le comportement fonctionnel du logiciel et sa conformité avec la spécification du logiciel, selon la technique de sélection des données de test en distingue quatre méthodes de test fonctionnelles [11] :

1. Analyse partitionnelle des domaines des données d'entrée et test aux limites
2. Le test combinatoire
3. Le test aléatoire
4. Test base modèle (model based testing ou MBT)

Dans les techniques de test traditionnel, le testeur a comme tâche d'étudier le système puis d'écrire et d'exécuter les différents scénarios de test. Ces scénarios (cas de test) sont écrits individuellement et peuvent être exécutés manuellement ou par certain nombre d'outils de « capture/play-back ».

En premier lieu, ces techniques de tests traditionnels souffrent beaucoup du « paradoxe des pesticides » dans lequel les tests deviennent de moins en moins utiles aux capture de bogues, car les bogues pour lesquelles ils ont été prévus sont pris en considération et évitées par les programmeur.

En second lieu, les scénarios de tests écrits à la main sont statiques et difficiles à modifier, mais le logiciel à l'essai évolue dynamiquement pendant que des fonctions sont ajoutées et changées. Quand les nouveaux dispositifs changent l'aspect et le comportement du logiciel existant, les essais doivent être modifiés pour s'y adapter. S'il est difficile de mettre à jour les essais, il sera difficile de justifier les coûts de maintenance d'essai.

Le MBT (model-based testing) est une technique de génération automatique des cas de test à partir de la modélisation du comportement du système sous le test. Les modèles sont utilisés pour la compréhension et la spécification des systèmes complexes dans plusieurs disciplines d'ingénierie (Fig. 1.7). L'utilisation des modèles pour la génération des cas de test à partir de modèles est motivée par de nombreux avantages [49].

- Les modèles fournissent une manière formelle de documentation, partage et réutilisation concernant le comportement du logiciel.
- Les modèles peuvent être introduits dans un générateur de cas de test pour la production automatique des cas de tests. Les modèles offrent ainsi le potentiel de développer des cas de test plus rapidement que par les méthodes de génération manuelles.
- Les algorithmes de génération pour les MBT peuvent garantir l'assurance basée sur les contraintes et sur les critères spécifiés par les développeurs de test.
- Les processus de création du modèle d'un logiciel peut augmenter la compréhension du logiciel par les testeurs ce qui peut mener à une amélioration des cas de test généré même si elle n'est pas faites par un générateur automatique des cas de test.

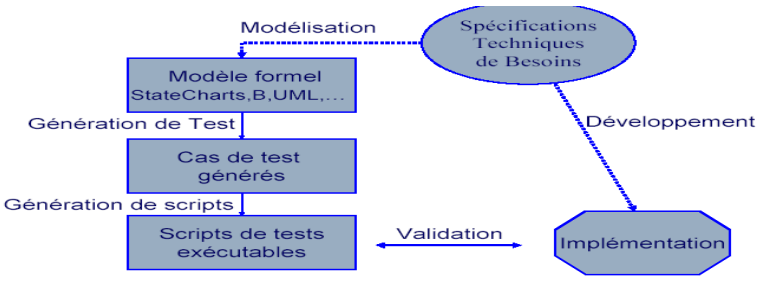


Figure 1.7 : Processus du test basé–modèle

Le test fonctionnel présente les avantages suivants :

1. Test très tôt dés que la spécification devient disponible.
2. Permet d'écrire les tests avant le codage.

Il souffre des inconvénients suivants :

1. L'efficacité du test repose sur la qualité de la spécification (doit être complète et compréhensible pour le concepteur des tests).
2. Le danger est l'explosion combinatoire qu'entraîne un grand nombre d'entrées du programme.

8.2.3 Le test aléatoire

Par définition, la sélection est effectuée en choisissant au hasard un jeu de tests parmi les entrées acceptable du logiciel. Les méthodes de sélection de jeux de tests aléatoires sont donc à priori très simple, sous réserve de disposer d'une caractérisation de l'ensemble des entrées acceptable du logiciel testé. Lorsque la loi de probabilité des entrées lors de l'utilisation effective du produit logiciel est connue, elle peut être utilisée pour guider la sélection. De plus, il est possible de définir un modèle de croissance de fiabilité reposant sur cette loi, ce dernier permet alors de fournir un critère d'arrêt du test. Que la loi de probabilité utilisée soit uniforme ou dépendante des utilisations futures, on parle indifféremment de méthodes de sélection de jeux de tests aléatoires ou statiques [7].

9 Complémentarité test fonctionnel – test structurel

Les tests fonctionnels et les tests structurels sont complémentaires et permettent de mettre en évidence différents défauts du programme. Les tests fonctionnels permettent principalement de détecter des défauts dus à une mauvaise compréhension des spécifications du logiciel. Les tests structurels permettent de détecter les défauts liés à la programmation [13].

Remarque

Il existe un autre type de test le test mixte (boite grise). Contrairement à ce que peut laisser croire cette terminologie, le *test boite grise* n'est pas une méthode de test intermédiaire entre le test « boite blanche » et le test « boite noire » mais plutôt une union des deux. L'intérêt du test mixte consiste à utiliser le test fonctionnel pour pallier aux inconvénients du test structurel et inversement.

10 Difficulté du test

Le test exhaustif est en général impossible à réaliser. En test fonctionnel l'ensemble des données est en général infini ou de très grande taille. Par exemple, un logiciel avec 5 entrées sur 8 bits admet 1280 valeurs différentes en entrée. En test structurel, le parcours du graphe de flot de contrôle conduit à une forte explosion combinatoire.

En d'autres termes : ***Le test ne peut être qu'une méthode de vérification partielle de logiciels et la qualité du test dépend de la pertinence du choix de données de test.***

Il existe aussi des difficultés d'ordre psychologique ou «culturel». En effet, le test est un processus destructif : un bon test est un test qui trouve une erreur alors que l'activité de programmation est un processus constructif où on cherche à établir des résultats corrects.

Les erreurs peuvent être dues à des incompréhensions des spécifications ou de mauvais choix d'implantation.

En d'autres termes : ***L'activité de test s'inscrit dans le contrôle de qualité et est indépendante du développement.***

L'exécution des tests peut être automatisée on peut écrire des programmes qui permettent d'enchaîner l'exécution de plusieurs tests, et qui vérifient que le résultat est correct. On peut ainsi facilement faire des tests de *non régression*, c'est-à-dire réexécuter un ensemble de tests et ainsi vérifier que les fonctionnalités déjà testées continuent à produire les résultats attendus [48].

Dans la pratique industrielle du test, l'automatisation est trop souvent limitée à l'exécution des tests, parfois avec une mesure de la couverture, et à la comparaison des résultats obtenus avec ceux attendus. C'est à l'utilisateur d'élaborer les cas à tester et les données de test (entrées et sorties attendues), éventuellement à l'aide d'outils de préparation de "scripts de test". Par conséquent, le nombre de tests effectués est limité par le coût et la disponibilité des ingénieurs compétents et la couverture des tests peuvent être difficiles à évaluer. Or, la génération automatique des données de test et d'un oracle permet une sélection des cas de test basée sur des critères justifiés par des hypothèses explicite [46].

L'état actuel de la recherche dans le domaine du test logiciel permet de dégager les méthodes largement utilisées. On peut décomposer le problème en trois étapes principales (voir la figure 1.8).

1. La génération automatique des tests, qui peut se faire, soit à partir du code du programme sous test et on parle alors de test structurel, soit à partir d'une spécification formelle du programme sous test et on parle alors de test fonctionnel.
2. La soumission des tests au programme sous test.
3. L'analyse automatique des résultats de test qui est effectuée par l'oracle.

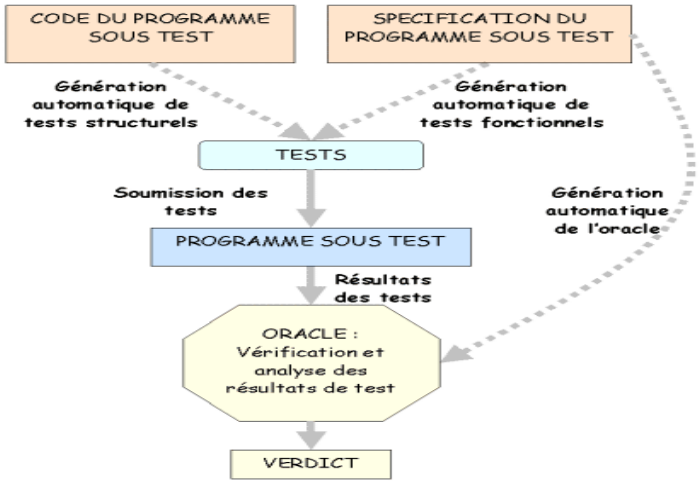


Figure 1.8 : Génération automatique des tests dynamiques

A titre d'exemple, Junit est un ensemble de classes Java réutilisables, qui permet d'enchaîner l'exécution de tests de programmes Java. Junit permet en particulier d'écrire des tests unitaires ; de lancer automatiquement l'exécution d'un ensemble de tests ; d'effectuer facilement des tests de non régression.

Conclusion

D'après ce qu'on a vu dans ce chapitre, on peut conclure que le test est très important pour la vérification de la correction d'un logiciel. Les méthodes de test peuvent être statiques ou dynamiques, dans les méthodes dynamiques on peut trouver des techniques fonctionnelles et structurelles d'une façon complémentaire rendant ainsi le test plus efficace. Le test peut être appliqué sur des procédures, des méthodes, des objets, ou sur des aspects. Dans le chapitre suivant nous aborderons la notion d'aspect qui constitue le contexte de notre travail.

CHAPITRE 2
LA PROGRAMMATION ORIENTÉE ASPECT ET ASPECTJ

Tout système informatique assure différents besoins fonctionnels, qui font son comportement et constituent l'objectif pour lequel il a été développé, et des besoins non-fonctionnels qui décrivent des contraintes imposées pour la satisfaction des besoins fonctionnels et jouent le rôle de support aux fonctionnalités premières d'un système.

L'implémentation des besoins non-fonctionnels pose souvent des problèmes avec les méthodes de programmation actuelles car il est difficile d'en limiter la portée dans un module bien circonscrit, et en se retrouve avec des problématiques dont l'implémentation est dispersée dans les différents modules fonctionnels du système. L'approches orientée aspects constituent un nouveau paradigme qui vient combler l'absence d'un support adéquat des besoins non-fonctionnels. Ce chapitre introduit l'approche orientée aspects, ses principes et ses concepts et situe son apport face à l'approche orientée objets.

1 Pourquoi la POA ?

Dans la programmation structurelle, procédurale et orienté-objet, l'implémentation des besoins non-fonctionnels tels que la sécurité ou la gestion des transactions est dispersée sur plusieurs modules. Ce qui conduit à dupliquer des parties du code dans les différents modules fonctionnels du système. Autrement dit, les besoins non-fonctionnels ne bénéficient pas d'une encapsulation adéquate ni au niveau des modèles de conception ni au niveau des langages de programmation. Cette situation est désignée par l'entrecoupage des préoccupations «crosscutting concern » ce qui rend difficile la lecture du code source ainsi que sa maintenance. La programmation orienté aspect tente de résoudre les problèmes des approches actuelles en séparant les préoccupations d'une application moyennant des concepts spécifiques.

2 Problèmes des approches classiques

2.1 Problème du couplage des préoccupations

Dans les méthodes de programmation (structurelles, procédurales ou orientées objets), il est presque impossible d'éliminer le couplage qui existe entre les frameworks (l'implémentation des besoins non fonctionnels) et les couches applicatives (l'implémentation des besoins fonctionnels). Par exemple, les frameworks de traçage et de sécurité sont utilisés dans tous les niveaux de l'application. Que se soit par l'appel d'une simple méthode ou l'ajout d'un bloc de code, ce couplage semble inéluctable. De ce fait, pour la couche métier (applicative), on trouve du code de gestion de sécurité et de traçage à coté et à l'intérieur des règles métier. La couche métier est alors dépendante des frameworks du traçage et de sécurité [35]. La figure 2.1 illustre ce problème de dépendance.

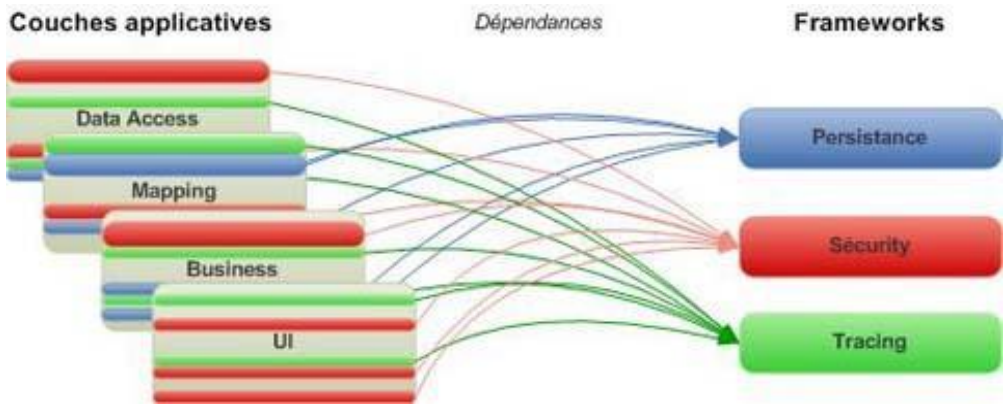


Figure 2.1 : Problème du couplage des frameworks avec les couches applicatives

Cette forte dépendance pose deux problèmes principaux :

1. L'Eparpillement : l'implémentation des besoins non fonctionnels est dispersée dans plusieurs modules
2. L'Enchevêtrement : un même module renferme des éléments de plusieurs problématiques

Ces deux problèmes entraînent des conséquences négatives. Citons entre autre :

a. Traçage difficile. Les différentes préoccupations d'un logiciel deviennent difficilement identifiables dans l'implémentation. Il en résulte une correspondance assez obscure entre les besoins et leurs implémentations [3].

b. Diminution de la productivité. La prise en considération de plusieurs besoins au sein d'un même module empêche le programmeur de se focaliser uniquement sur son but premier. Le danger d'accorder trop ou pas assez d'importance aux aspects accessoires d'un module en découle directement [3].

c. Diminution de la réutilisation du code. Dans les conditions actuelles, un module implémente de multiples besoins. D'autre systèmes nécessitant des fonctionnalités similaires pourraient ne pas pouvoir réutiliser le module tel quel, entraînant de nouveau une diminution de la productivité à moyen terme [3].

d. Diminution de la qualité du code. Les programmeurs ne peuvent pas se concentrer sur toutes les contraintes à la fois. L'implémentation disparate de certaines préoccupations peut entraîner des effets de bord non désirés [3].

e. Maintenance et évolutivité du code difficile. Lorsqu'on souhaite faire évoluer le système, on doit modifier de nombreux modules. Modifier chaque sous-système pour effectuer les modifications souhaitées peut conduire à des incohérences [3].

2.2 Les solutions

La mise en évidence des problèmes qui affectent les systèmes actuels et leur développement a engendré de nombreuses solutions. Nous citons :

a- Les patrons de conception (design patterns)

Ces modèles de programmation s'appuient sur la conception objet pour répondre a un certain nombre de problèmes récurrents dans le développement logiciel.

Tout un lot de patterns est ainsi apparu permettant tout d'abord d'accroître la productivité mais, également, la compréhension des architectures logicielles. En effet, une fois le pattern localisé, sa simple dénomination permet à un développeur de comprendre le comportement intrinsèque de la solution mise en place. Le code des patterns proposés est toujours implémenté plus ou moins de la même manière et permet une lisibilité accrue du code et donc une meilleure maintenance.

Malheureusement les dépendances créées entre les objets réalisant ce pattern subsistent fortement et ne permettent pas de réutiliser les fonctionnalités métiers développées [3].

Cette approche apporte une solution très élégante aux problèmes récurrents mais elle reste toutefois complexe à appréhender ou à mettre en œuvre pour certains profils. Cela peut être du à un problème de conceptualisation. Les concepts tels que les classes et les interfaces, l'héritage et le polymorphisme paraissent déjà un peu abstraits à certains. Il semble donc normal que remonter d'un niveau d'abstraction, ce que fait la notion de pattern, ne simplifie pas les choses [9].

b- Introspection / réflexion. La réflexion et l'introspection permettent à des objets de s'analyser ou de se définir eux-mêmes. En java ces fonctionnalités sont fournies par la classe et tous les objets du paquetage java.lang.reflect.

Les solutions obtenues grâces à ces principes permettent de rendre les objets fortement indépendants les uns des autres mais complexifie généralement fortement le code obtenu. On se retrouve donc à résoudre les problèmes de qualité liés aux dépendances entre les objets en dégradant la lisibilité et la maintenabilité du code.

Beaucoup de solutions utilisent ces fonctionnalités, notamment dans les bibliothèques de programmation. Ainsi la solution JBDC permet aux développeurs de se rendre indépendant du support des données. La complexité réside dans le code interne à la bibliothèque, loin des développeurs qui l'utilisent [3].

c- Model Driven Architecture (MDA). Le développement orienté modèle propose une montée en abstraction de la conception des applications en se fondant sur la modélisation via UML. Le but est d'arriver à un modèle général qui représentera la solution donnée au problème métiers uniquement, indépendamment de toute spécification technique.

Ce modèle sera ensuite transformé pour obtenir un modèle dépendant d'une plateforme cible pour ensuite produire le code technique dans les langages souhaités.

Cette approche rencontre encore beaucoup de difficultés notamment dans la transformation des modèles et dans le maintien de la cohérence entre le code et le modèle de départ. En effet, une fois le code produit, il est difficile de répercuter toutes les modifications faites dans les modèles et vice-versa.

D'autre part, le niveau et l'étendue des connaissances à acquérir pour maîtriser entièrement ce type de développement sont un frein à son utilisation.

Les résultats sont pourtant très prometteurs et certaines entreprises mettent déjà en œuvre des chaînes de développement basées sur ces principes. Il est possible de penser que le développement dirigé par les modèles vienne se greffer à la programmation par aspect ou le contraire dans les prochaines années [3].

d- L'approche par composants. L'approche du développement par composant prend en compte un certain nombre de préoccupations transverses que l'on rencontre souvent. La solution proposée étant de plonger les objets métiers au sein de ces composants qui se chargent alors de résoudre les préoccupations transverses.

A ce jour des solutions comme EJB implémentent ces principes et offrent aux développeurs un moyen d'arriver à de bons résultats. Toutefois, l'ensemble des préoccupations est fixé et normé par l'OMG (Object Management Group) et il très difficile de définir de nouvelles préoccupations transverses [3].

e- L'approche orientée-aspect. La meilleure solution du problème d'enchevêtrement est de bien séparer les frameworks (l'implémentation des besoins non-fonctionnels) des couches applicatives (les besoins fonctionnels ou code de base, Fig. 2.2).

La programmation orientée-aspects permet d'implémenter chaque problématique indépendamment des autres, puis, de les assembler selon des règles bien définies. Donc elle promet une meilleure productivité, une meilleure réutilisation du code et une meilleure adaptation du code aux changements [9].

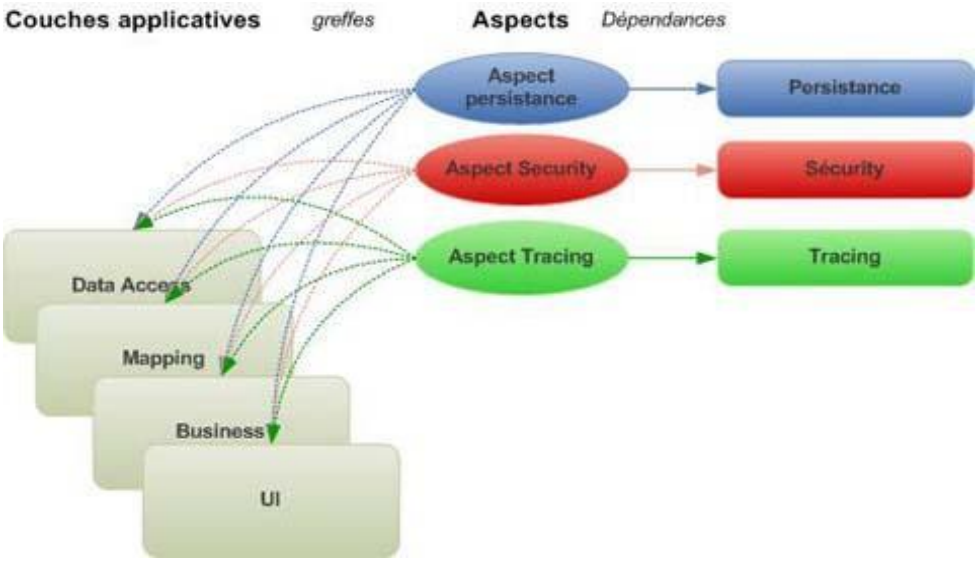


Figure 2.2 : Séparation des frameworks des couches applicatives

3 Principe de la séparation des préoccupations

La loi de Demeter, datant de 1987, s'annonce comme suit :

« L'efficacité d'un projet informatique augmente si toutes les préoccupations de nature différentes sont bien modularisées et si un programmeur qui désire faire une modification ne doit parler qu'à ses voisins directs pour la faire tout en étant sur de ne pas introduire de bugs ».

La séparation des préoccupations met en œuvre la loi de Demeter moyennant des concepts ad hoc que nous expliquons ci-après.

La séparation des préoccupations est un processus qui décompose et divise un problème (programme) en modules distincts où chacun renferme une seule préoccupation [6].

Tout système informatique peut être vu comme un ensemble de préoccupations. Une préoccupation est en fait un but particulier, une problématique d'un logiciel qui correspond à l'une de ses exigences. Dans la plupart des implémentations actuelles, on procède à ce qu'on appelle la découpe fonctionnelle du système, c'est-à-dire que l'on divise le système en modules, représentant chacun une fonctionnalité particulière. les exigences non-fonctionnelles sont difficilement prises en compte dans une telle découpe et on se contente donc de les intégrer dans les différents modules fonctionnels [3, 41].

Voici quelques exemples de grandes catégories de préoccupations :

Préoccupations orientées données

1. Persistance des données
2. Contrôle d'accès aux données (gestion de droits, d'utilisateurs autorisés, de groupes)

Préoccupations orientées fonctionnalités

1. Fonction de calcul de la TVA dans une application comptable
2. Fonction de calcul de charge salariale dans une application de gestion du personnel

Préoccupations orientées règles de gestion

1. Règles organisationnelles dans une application de gestion du personnel
2. Règles de workflow dans le suivi du cheminement de l'information

Préoccupations architecturales

1. Interopérabilité entre applications hétérogènes
2. Schémas de connexion entre composants répartis

Préoccupation de débogage

1. Génération de la trace

4 La programmation orientée-aspects

La POA (Aspect-Oriented Programming, Programmation Orientée-Aspects) est une approche particulière de séparation des préoccupations. Elle a été définie par Gregor Kickzales et son équipe (du laboratoire de recherche PARC de Xerox) en 1996. Tout comme la programmation structurée et la programmation orientée objets ont, dans leur temps, introduit de nouvelles approches pour concevoir les programmes et un ensemble de règles et de conventions pour aider les programmeurs à produire du code plus lisible et plus réutilisable (éradication des "gotos" au profit des boucles, introduction des types et des structurations modulaires autour des classes), la POA est une philosophie de programmation qui est essentiellement une question de style. La POA résout donc des problèmes qui peuvent être traités dans des approches classiques, d'une manière élégante [33].

La POA se définie formellement comme une technique de conception et de programmation qui vient en complément de l'approche Orientée Objet ou procédurale. Elle permet de factoriser (et donc de rendre plus claires) certaines fonctionnalités, dont l'implémentation aurait nécessairement été répartie sur plusieurs classes et méthodes dans le monde objet ou sur plusieurs bibliothèques et fonctions dans le monde procédural. La POA n'est pas une technique autonome de conception ou de programmation. Sans code procédural ou objet, la notion d'Aspect perd tout son sens. Inversement, on pourrait dire que la programmation orientée objet ou procédurale n'est pas complète puisque inaptes à mettre en facteur ou bien séparer certaines responsabilités des éléments logiciels [9].

La POA peut aussi se définir comme une utilisation particulière de l'instrumentation de code qui s'attache essentiellement à résoudre des problèmes de conception. La POA ne vise donc pas, a priori, à satisfaire les besoins de génération automatique de code ou de documentation [9].

4.1 Motivations de la POA

Pour les langages orientés-objets l'unité naturelle de la modularité est la classe. Mais dans ce genre de langage, les préoccupations ayant une portée globale sont difficilement traitables au niveau des classes car elles concernent précisément plusieurs classes, et donc seraient non réutilisables. Elles ne pourraient pas être raffinées ou dérivées, elles s'étendent sur l'ensemble du programme d'une façon inorganisée, et en résumé, elles sont difficiles à traiter.

La programmation orientée-aspects propose une organisation différente tant pour les éléments de code que les équipes de développement. Elle permette une bonne répartition des compétences, une certaine robustesse et une bonne performance des applications sans induire trop de redondance.

La POA est la prise en compte du fait qu'il existe un certain nombre de considérations qui ne sont pas bien traitées par les méthodologies traditionnelles de programmation. Par exemple si on prend le problème de l'implémentation d'une politique de sécurité dans une application, on constate tout de suite que par sa nature, celle-ci s'étend au delà des frontières d'un module particulier de l'application. Plus encore, la politique de la sécurité doit être uniformément appliquée à tout ajout dû à l'évolution de l'application. De plus, la politique de la sécurité est amenée à évoluer elle-même. Avec les langages de programmation traditionnels il est difficile d'assurer ce qui vient d'être décrit.

4.2 Avantages et caractéristiques de la programmation orientée-aspects

Aidée par une compatibilité entre ses concepts sous-jacents et ceux existants, la POA peut être intégrée à moindre coût par les entreprises en étendant leurs outils. Par exemple, la POA pour JAVA est intégrée dans Eclipse via un plugin qui permet l'utilisation du langage AspectJ.

La compatibilité entre les concepts utilisés par la POA et les outils existants est due au fait que la POA ne remet pas en cause les autres paradigmes de programmation (comme l'approche procédurale ou objet). Au contraire, elle les étend en proposant des mécanismes complémentaires afin d'améliorer la modularité d'une application (et donc faciliter la réutilisation et la maintenance).

La POA repose en partie sur des techniques existantes telles que la méta programmation ou encore la réflexivité et elle se positionne en tant que digne successeur de la POO, sans pour autant remplacer celle-ci puisqu'elle ne fait qu'étendre les concepts existants [31].

5 Modèle de la POA

Le modèle de la POA fonde la définition d'une application sur un programme principal, appelé le programme de référence, et un ensemble d'aspects indépendants. Le programme de référence détermine la sémantique métier de l'application. Les aspects sont tissés à ce programme afin de l'étendre ou de l'adapter pour un usage particulier.

5.1 Modèle général de la POA

Le programme de référence et les aspects sont, en théorie, développés séparément. Les aspects sont ensuite, tissés au programme de référence à l'aide d'un outil spécifique, le tisseur d'aspects (aspect weaver). Ce dernier produit un nouveau programme qui contient les fonctionnalités de référence et celles fournies par les aspects (Fig 2.3).

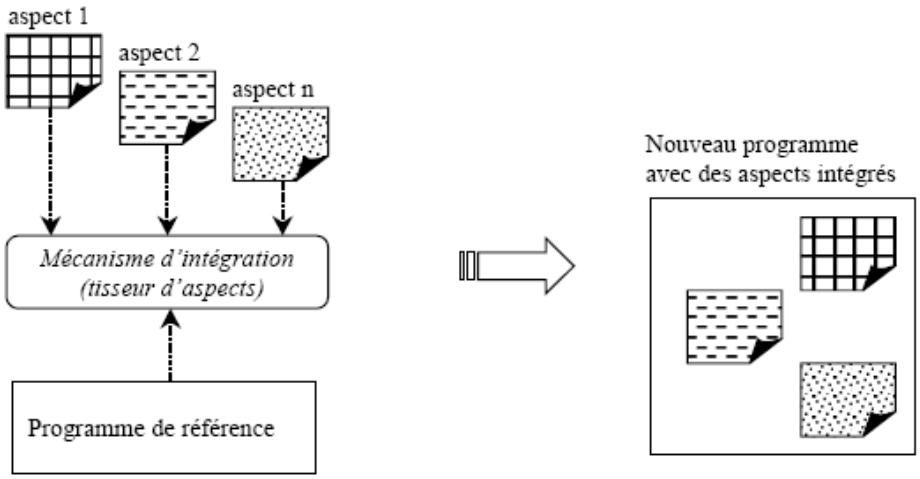


Figure 2.3 : Modèle général de la POA

Le programme de référence est défini dans un langage de programmation traditionnel (un langage à objets, par exemple). Il sert de référentiel pour introduire les points servant à ancrer les aspects. Il peut ainsi être considéré comme étant l'environnement ou bien le contexte d'exécution des aspects attachés. Puisque les unités de composition sont des aspects, nous regardons ci-après en détails les mécanismes proposés par la POA pour :

1. Définir les aspects,
2. Composer les aspects spécifier le moment où on réalise effectivement la composition,
3. Coupler les aspects et le programme de référence,
4. Faire évoluer et adapter le programme qui résulte de la composition des aspects au sein du programme de référence.

En schématisant, on peut dire que pour la POA, un aspect est caractérisé par deux éléments (1) le code ajouté et (2) la localisation

Le code ajouté constitue ce que l'aspect apporte au programme de référence. Ceci est appelé également l'action ou bien la méthode de l'aspect.

La localisation exprime l'endroit, dans le programme de référence, où l'aspect doit être attaché.

Puisque les aspects représentent la sémantique additionnelle au programme de référence, les aspects ne peuvent pas s'exécuter indépendamment du programme de référence. En plus, il est nécessaire d'avoir un mécanisme permettant de passer des paramètres entre le Programme de référence et des aspects. AspectJ propose de s'appuyer sur les paramètres définis dans les points de jonction. Ce mécanisme n'est pas très intuitif, mais s'avère Intéressant et relativement performant.

5.2 Mécanisme de composition

A propos du mécanisme de composition de la POA, nous distinguons deux facettes principales :

a- La composition des aspects au sein du programme de référence. A ce propos, nous pouvons dire que la POA est une approche de bas niveau. Ceci, parce qu'elle travaille directement au niveau du code source. Aucune vision de haut niveau d'abstraction n'est utilisée pour spécifier les aspects ainsi que le programme de référence. Nous considérons une telle composition comme une intégration. La structure interne du programme de référence est complètement visible aux aspects. Par conséquent, il est difficile de faire évoluer le programme de référence car un changement mineur de ce programme peut perturber la localisation des endroits où il faut attacher les aspects. Ceci peut être vu clairement dans AspectJ où les méthodes du programme de référence sont utilisées pour définir les endroits où attacher les aspects.

b- La composition des aspects attachés à la même localisation dans le programme de référence. A ce propos, nous pouvons distinguer deux manières d'opérer :

- Par ordonnancement des aspects, on ne peut cependant pas avoir un contrôle très fin sur la composition des aspects.
- Par le tissage, la composition d'aspects peut être réalisée en entremêlant les instructions de ces aspects. Ceci permet un contrôle très fin sur la coordination des aspects.

5.3 Mécanisme d'implémentation

Le mécanisme d'implémentation fixe le moment où l'intégration des aspects au sein du programme de référence, doit être accomplie. Deux politiques peuvent être utilisées :

- L'intégration statique consiste à utiliser un compilateur qui génère le nouveau programme en insérant les aspects au programme de référence. De cette manière, le nouveau programme contient les aspects intégrés qui sont exprimés en termes d'instructions additionnelles dans les méthodes du programme de référence. Cette politique offre une exécution performante mais peu flexible. Elle peut travailler en manipulant le code source et même en code binaire (bytecodes, dans le cas de Java) du programme de référence.
- L'intégration dynamique consiste à utiliser un interpréteur pour ajouter dynamiquement, lors de l'exécution, les aspects au

programme de référence. Pour se faire, l'interpréteur doit instrumenter l'exécution du programme de référence afin de capturer les points où il faut ajouter les aspects et ensuite, exécuter ces aspects. Il est clair que l'intégration dynamique est plus flexible que l'intégration statique puisque les aspects peuvent être ajoutés/enlevés à tout moment. Cependant, le temps d'exécution est plus important. L'intégration dynamique peut être appliquée au code binaire du programme de référence.

6 La comparaison entre la POA et la POO

Si une classe est une responsabilité, nous pouvons dire qu'un aspect est une fonctionnalité. Un aspect peut, par exemple, contenir les fonctionnalités de traces, de sécurité ou encore de persistance des données d'un programme. Dans cette optique, nous pouvons dire que la POA réduit la dispersion de code d'un programme [36] comme illustré dans le schéma suivant.

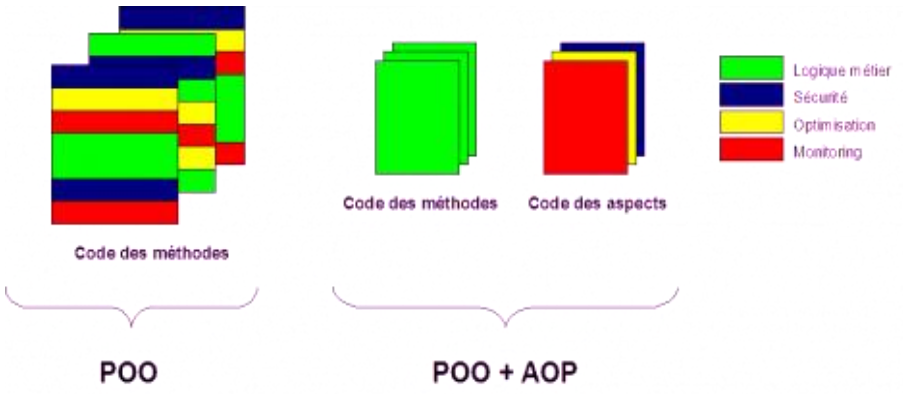


Figure 2.4 : Comparaison entre La POA et la POO

6.1 La place de la POA au sein de la POO

Nous pouvons dire que la programmation orientée-objets correspond à une programmation verticale. Cela veut dire que nous structurons les fonctionnalités de manière hiérarchique. C'est d'autant plus évident lorsque nous représentons un programme orienté-objets en UML.

Par analogie, la programmation orientée aspect serait, quant à elle, plutôt une programmation horizontale, opérant par couches. Elle va implémenter les fonctionnalités transversales du programme.

Les schémas suivant illustre très bien la place de la programmation orientée-aspects au sein de la programmation orientée objets. Les dépendances entre objets sont extériorisées par les aspects [36].

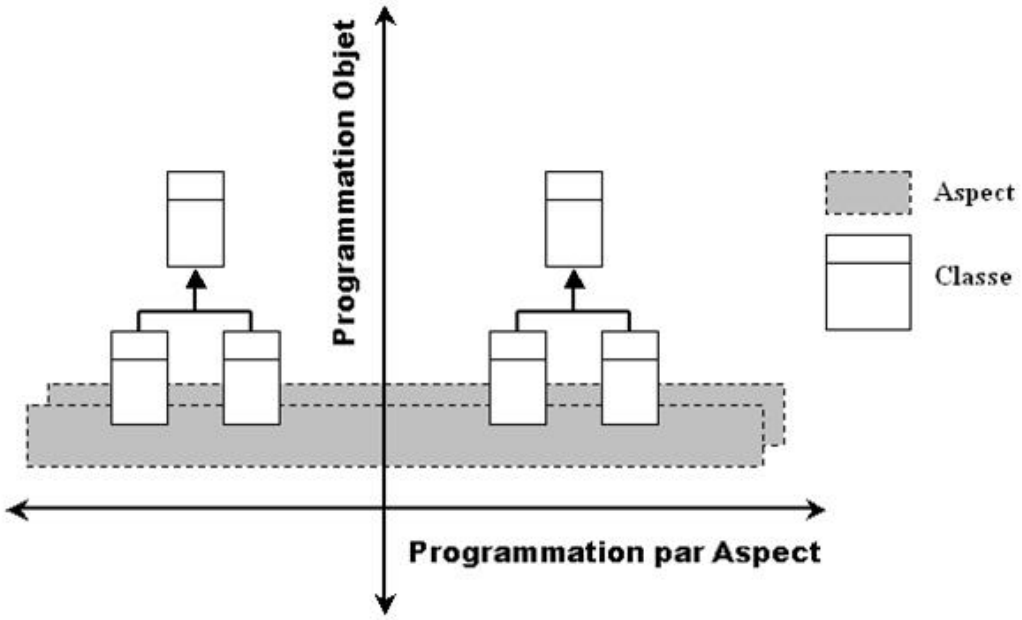


Figure 2.5 : La place de la POA au sein de la POO

6.2 Les dépendances entre le code orienté objet et les services

La programmation orientée aspect possède une autre force qui est l'inversion de dépendance. Effectivement, nous pouvons ne faire dépendre que les aspects d'un service donné. De ce fait, nous réduisons les risques de mauvaises utilisations de ces services, mais aussi la dépendance de l'ensemble du programme sur celui-ci. Il sera possible de changer ce service en modifiant uniquement les aspects du programme.

Dans le schéma suivant, il est bien illustré l'existence une dépendance entre les services et le programme codé en orientée-objets (dessin de gauche). Par contre, il n'existe aucune dépendance entre les aspects et le programme pour le programme codé en orientée aspect (à droite), et la dépendance avec les services n'existe que sur les aspects. C'est ce qui illustre l'inversion de dépendance [36].

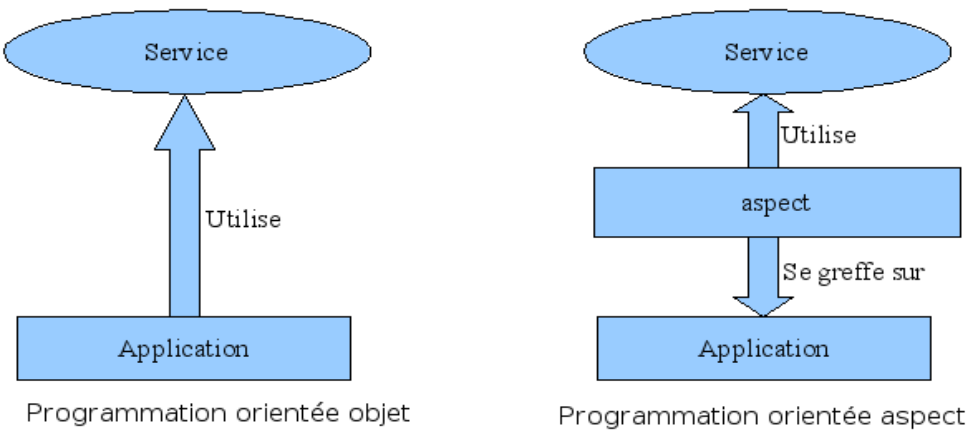


Figure 2.6 : Les dépendances entre le code orienté objet et les services

7 Le langage AspectJ

Plusieurs langages ont été développés, AspectJ est le langage le plus mature. Les premières versions datent de 1998. Il étend le langage Java en introduisant de nouveaux mots- clés permettant de programmer des aspects. Mais AspectJ n'est pas l'unique langage orienté-aspects. Même si celui-ci reste le plus utilisé, d'autres outils existent. On peut notamment citer JAC, JBoss AOP et AspectWerkz rien que pour le langage Java, mais il existe également des outils pour les langages C (nommé AspectC), C++ (nommé AspectC++), C# ou Smalltalk [12].

Contrairement à ses concurrents Jboss AOP et JAC, AspectJ n'est pas un framework de travail mais il est basé sur des extensions au langage java.

Dans AspectJ, l'unité n'est pas la classe, mais une préoccupation (aspect), qui se partage entre plusieurs classes. Les aspects peuvent être des propriétés, des zones d'intérêt, d'un système et la POA décrit leurs relations, les compose ensemble dans un programme. Les aspects encapsulent un comportement commun à plusieurs classes. Les aspects d'un système peuvent être insérés, changés, supprimés pendant la compilation [51].

7.1 Pourquoi utiliser AspectJ?

AspectJ présente divers avantages :

- L'utilisation d'AspectJ peut réduire considérablement la taille d'un programme Java sans perte de performance, et simplifie la conception d'autant.
- Améliore la modularité et la réutilisabilité du code.
- Particulièrement utile pour déboguer de grands projets.

7.2 Caractéristiques d'AspectJ

Nous pouvons résumer les caractéristiques d'AspectJ comme suit :

- Supplément à Java qui peut être ajouté à un programme existant (et remplacer un tas de code ou lui ajouter des contrôles ou traitement de débogage).
- Les points de jointures représentent des points définis dans l'exécution d'un programme, tels l'appel d'une méthode, la réalisation d'une exception, et ainsi de suite.
- Un point de jointure peut contenir d'autres points de jointures.

- Un point de coupure est un constructeur qui comprend un ensemble de points de jointure, selon un critère donné. Par exemple l'ensemble des méthodes à l'exécution d'un package.
- L'advice est le code qui est exécuté avant, après, ou pendant un point de jointure.
- Le compilateur AJC (AspectJ compiler) génère à partir d'un programme source AspectJ (soit un fichier source ***.java***) un fichier bytecode ***.class***.
- Le code AspectJ peut être réutilisé dans d'autres programmes.

7.3 Aperçu technique sur le langage AspectJ

Un aspect est une entité définissant un concept transversal aux objets métiers de l'application comme la persistance ou la journalisation. Il est composé de point de coupure définissant les points d'exécution du programme qui seront interceptés par les advices comportant le code à exécuter par rapport à un point de coupure et de déclarations inter-types qui permettent d'ajouter des éléments aux classes métier (attribut, méthode, interface implémentée, ...).

Tout programme Java est compatible à AspectJ. Toutefois, AspectJ permet également au programmeur de définir des constructions spéciales nommées "aspects" [12].

a- Définir un aspect. Définir un nouvel aspect est très semblable à création d'une nouvelle classe. Le mot-clé *class* est juste remplacé par *aspect*. L'aspect est vide et ne définit pas un nouveau comportement. Dans un aspect nous allons pouvoir définir des coupes et des codes advice qui induiront les fonctionnalités liées à cet aspect [12].

Par exemple, un aspect qui se charge d'afficher « Hello Word !» lorsque la fonction main est appelée est le suivant :

```
public aspect HelloWorld {

  pointcut mainCall():  //Coupe
  call (public static void main(String [] args)));

  before () mainCall(){  //Code Advice

    System.out.println("Hello world!");

  }

}
```

b- Les points de jonction (joinpoint ou point de jointure). Un ensemble de points de jonction est identifié dans une coupe à l'aide du mot-clé *pointcut*. Une expression est également fournie pour filtrer l'ensemble obtenu [12].

Un point de jonction représente un événement dans l'exécution du programme qui peut être intercepté pour exécuter un code *advice* correspondant.

Plusieurs types d'événements peuvent faire figure de points de jonction.

- **Méthodes**. C'est autour des méthodes que les aspects se greffent le plus souvent. Ce n'est pas étonnant puisque les méthodes forment l'outil principal de la POO et structurent l'exécution du programme. Les événements liés aux méthodes qui constituent des points de jonction sont l'appel d'une méthode et l'exécution de celle-ci. La spécification de ces points se fait dans l'aspect par ***call*** *(methexpr)* ou ***execution*** *(methexpr)* qui identifient tous les appels vers des méthodes dont le profil correspond à une description donnée par *methexpr*.
- **Constructeurs.** Les constructeurs peuvent être considérés comme des méthodes particulières. Il s'agit de méthodes invoquées lorsqu'un objet est instancié. Comme pour les méthodes, la POA permet d'intercepter cet événement. Il est intéressant de remarquer que l'on peut intercepter les appels des constructeurs avec le mot-clé ***call*** que nous avons explicité précédemment. Il suffit de fournir une *methexpr* identifiant les méthodes de nom *new*.
- **Exceptions.** Les événements de levée et de récupération d'exceptions peuvent aussi constituer des points de jonction. Le code à exécuter lors de la levée de cette exception sera défini une seule fois dans un aspect. La spécification de ces points se fait dans l'aspect par ***handler*** *(exceptexpr)* qui identifie toutes les récupérations d'exception dont le profil vérifie *exceptexpr*. Il s'agit donc, en Java de l'exécution des blocs *catch*. AspectJ ne permet pour le moment que l'utilisation de codes advice de type *before* sur les points de jonction de récupération d'exception.
- **Attributs.** La lecture et la modification d'attributs constituent également des points de jonction. On peut penser notamment à une utilisation à des fins de persistance. La spécification de ces points se fait dans l'aspect par ***get****(attrexpr)* qui identifie tous les points de jonction représentant la lecture d'un attribut dont le profil vérifie *attrexpr* et ***set****(attrexpr)* qui identifie toutes les modifications d'un attribut dont le profil est compatible

à *attrexpr*. Ces mots clés sont intéressants pour des aspects qui souhaitent intercepter la modification de l'état d'un objet [12].

Nous venons de voir certains types d'événements qui peuvent constituer des points de jonction. Mais la POA peut connaître des limites quant à la granularité correspondante aux points de jonction. En effet, les instructions (if, while, for, switch,…) sont jugées trop fines par certains outils de la POA et ne sont donc pas interceptables. C'est par exemple le cas pour le langage AspectJ [12].

c- Les coupes (pointcut ou point de coupure). Elles permettent au programmeur de spécifier les points de jonction comme l'appel à une méthode, l'instanciation d'objet ou l'accès à une variable). Tout "pointcut" est une expression vérifiant la correspondance d'un point de jonction [29]. Les points de coupure représentent des coupes qui sont de deux types : les coupes simples et les coupes composées.

Une coupe est introduite grâce au mot clé pointcut. La syntaxe est :

pointcut nomDeLaCoupe (paramètres) Définition de la coupe

Une coupe permet de regrouper un ou plusieurs points de jonction. Pour savoir quelle syntaxe de définition de coupe utiliser, il faut savoir sur quelle sorte de point de jonction on souhaite faire une coupe pour exécuter du code. Un point de jonction peut être une méthode. Le mot clé alors utilisé pour définir la coupe peut être *call* ou *execution*. Cela dépend si l'on souhaite exécuter le code advice lors de l'appel ou dans le code de la méthode exécutée. Pour résumer la différence entre *call* et *execution*, voici l'ordre d'exécution des codes advice liés [12].

1 - Code Advice de type « before » associé au point de jonction call.

2 - Code Advice de type « before » associé au point de jonction Exécution.

3 - Code de la méthode

4 - Code Advice de type « after » associé au point de jonction execution

5 - Code Advice de type « after » associé au point de jonction call.

Les déférents types de codes advice seront décrits plus loin dans ce chapitre.

Outre les méthodes, un point de jonction peut concerner une lecture ou une écriture d'un attribut de classe, la récupération d'une exception, l'exécution d'un constructeur hérité, l'exécution d'un bloc de codes statiques particulier ou encore l'exécution d'un code advice. Selon le type de point de jonction, le mot clé à utiliser pour définir la coupe est différent.

La syntaxe est résumée dans le tableau suivant (extrait de [12]).

Syntaxe	Description du point de jonction
call (methodExp)	Appel d'une méthode dont le nom vérifie methodeExp.
execution(methodExp)	Exécution d'une méthode dont le nom vérifie methodeExp.
get (attributExp)	Lecture d'un attribut dont le nom vérifie attributExpr. Exemple get (int Point.x)
set (attributExp)	Ecriture d'un attribut dont le nom vérifie attributExp.
handler (exceptionExp)	Exécution d'un bloc de récupération d'une exception (catch) dont le nom vérifie exceptionExp Exemple : handler (IOException+)
initialization (constanteExp)	Exécution d'un constructeur de classe dont le nom vérifie constanteExp Exemple : initialization (Customer.new (..))
preinitialization (constanteExp)	Exécution d'un constructeur hérité dont le nom vérifie constanteExp
staticinitialization (classeExp)	Exécution d'un bloc de code static dans une classe dont le nom vérifie classeExp Exemple staticinitialization (Point)
adviceexecution ()	Exécution d'un code advice

Tableau 2.1 : Tableau récapitulatif des points de jonction possibles et la syntaxe à utiliser pour définir la coupe

Pour décrire des coupes génériques il est possible d'utiliser des *wildcards* qui sont des caractères permettant de définir de manière plus vaste une méthode ou une classe, etc. Les wildcards existants sont récapitulés dans le tableau suivant [12].

Wildcard	Utilisation
*	Remplace un nom (de classe, de paquetage, de méthode, d'attribut, etc..) ou simplement une partie de nom. Il peut aussi remplacer un type de retour ou un paramètre. Il signifie « n'importe quel nom » ou « n'importe quel type ». Exemple pour une expression sur une méthode **public** *fr.umlv.*.test.*.start*(int, String,*)
..	Utilisé pour omettre les paramètres des méthodes ou le chemin complet des paquetages. Exemple pour une expression sur une méthode **public** void fr..Test.SetParams(..) Cette exemple signifie « toutes les méthodes publiques SetParam quel que soient leurs paramètres, retournant void et situées dans des classes T est situées dans n'importe quel sous paquetage de fr ».
+	Permet de définir n'importe quel sous-type d'une classe ou d'une interface Exemple pour une expression sur une méthode void fr.umlv.test.IMouseListener+.set*(..); Cet exemple signifie « toutes les méthodes commençant par set des classes implémentant l'interface IMouseListener.»

Tableau 2.2 : Liste des wildcard et leur signification

Il est également possible de généraliser des coupes via des opérateurs logiques, il s'agit de coupes composées. En effet, il est possible de combiner des points de jonction avec les opérateurs logiques.

Par exemple le point de coupure

pointcut allGetSet(): **call** (void *..set*(..)) **|| call** (**..get*());

permet de joindre le point de jonction défini par les deux *call*. Il est également possible de faire du filtrage grâce à des mots-clés spécifiques. Ces mots-clés permettent de préciser les points de jonction selon l'appelant, l'appelé, le flot de contrôle ou encore les arguments des fonctions. Associés à des opérateurs logiques, ces mots clés permettent de définir de manière très précise les points de jonction. En résumé les opérateurs logiques et les mots-clés existants sont les suivants [12].

Mot clé	Signification
&&	Et logique
||	Ou logique
!	Négation logique
if (ExpressionBooléenne)	Evaluation de l'expression booléenne ExpressionBooléenne. Exemple pointcut test() if (thisJoinPoint. getArgs().length ==1) &&execution (**.*(..));
withincode (methodeExpression)	Vrai lorsque le point de jonction est défini dans une méthode dont la signature vérifie methodeExpression
within (typeExpression)	Vrai si l'événement se passe pendant l'exécution d'un code embarqué par une classe définie par typeExpression (cela inclut les méthodes statiques). Permet essentiellement de limiter la portée d'un autre pointcut à une certaine classe.
this (typeExpression)	Vrai lorsque l'événement se passe à l'intérieur d'une instance de classe de type typeExpression
target (typeExpression)	Vrai lorsque le type de l'objet destination du point de jonction vérifie typeExpression
cflow (coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (y compris l'entrée et la sortie)
cflowbelow (coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (sauf pour l'entrée et la sortie)

Tableau 2.3 : Opérateurs logiques, mots-clefs de filtrage et leur signification

d- Code advice. Une fois la coupe définie, il faut écrire le code à exécuter au moment où l'événement décrit par la coupe est levé. Avant d'écrire ce code advice, il faut décider à quel moment exécuter le code : avant l'événement, après ou autour de l'événement.

- advice before (avant). Le mot-clé before permet d'exécuter du code avant d'entrer dans le code lié à l'événement (exemple appel de méthode, lecture d'attribut, etc...).
- advice after (après). Le mot-clé after permet quant à lui permet d'exécuter du code après l'événement.
- advice around (autour). Le mot-cle around permet soit de remplacer l'événement en lui même, soit d'exécuter du code avant et après le point de jonction. Pour exécuter le code de l'événement il faut utiliser le mot-clé *proceed* à l'intérieur du code advice. Si la méthode a un type de retour, alors il faut faire précéder le mot-clé around de *Object*. L'appel à *proceed* renvoie alors un type *Object* qu'il est possible de récupérer pour le renvoyer [12].

Exemple : Voici un exemple très simple permettant de tracer l'exécution de la méthode *getX*.

```
pointcut getXValue() execution (int *..Point.getX());
Object around () getXValue() {
  System.out.println("=>Entrée dans getX");
  Object ret =proceed();
  System.out.println("<=Sortie de la méthode getX");
  Return ret;
}
```

e- Déclaration inter-types. Permet au programmeur d'ajouter des méthodes, des champs ou des interfaces à des classes existantes depuis l'aspect. L'exemple suivant ajoute une méthode acceptVisitor à la classe Point [33].

```
aspect VisitAspect {
  void Point.acceptVisitor(Visitor v) {
    v.visit(this);
  }
}
```

Le format d'un code aspect en AspectJ est le suivant

```
public aspect xxxx {
  pointcut anyMethod()
       execution (* org.eclipse..
*(..));
  before()  anyMethod(){
... code.consigne..
  }
  after() anyMethod(){
... code cnsigne
  }
}
```

Conclusion

Nous avons présenté dans ce chapitre le paradigme de programmation orienté aspects et nous avons passé en revue ses principes, ses concepts et ses avantages comparés au paradigme orienté objets. Il ressort de cette présentation que ce paradigme, relativement nouveau, représente une évolution certaine dans le domaine de la modélisation et de la séparation avancée des préoccupations en général. Cependant, face à cette évolution, de nouveaux problèmes apparaissent aussi bien sur le plan de l'application des concepts du paradigme à différent niveaux du cycle de vie d'un logiciel, que sur le plan de l'implémentation, mais aussi sur le plan du test qui est notre contexte dans ce travail. Le chapitre suivant dresse un panorama des approches de test orientées-aspects et met l'accent sur les défis que le paradigme AOP pose pour les activités de test.

CHAPITRE 3
PANORAMA DES APPROCHES DE TEST ORIENTÉES-ASPECTS

Quelle que soit la technologie de programmation utilisée, l'un des principaux objectifs du développement est la production d'un logiciel de haute qualité. Les programmes orientés-aspects, comme tout autre programmes, doivent être testés et valider pour la qualité.

Les questions clés liées à la qualité d'un logiciel orienté-aspects comprennent : Comment peut-on savoir si nous avons testé suffisamment un programme orienté–aspects ? Comment peut-on savoir si les objectifs de qualité sont atteints ?

Ces questions n'ont pas de réponses simples à cause des caractéristiques de l'entité aspect [1] et de la nouvelle dimension que la séparation des préoccupations introduit. Dans ce chapitre on s'intéresse aux approches de test dédiées aux programmes orientés-aspects en détaillant les nouveaux types de défauts que ces approches tentent de détecter et qui constituent un véritable défi pour le test.

1 Caractéristique de l'entité « Aspect »

Une analyse d'un programme orienté-aspects laisse apparaître certaines caractéristiques qu'il convient de considérer avec soin. Nous énumérons ce qui suit :

- Les aspects ne sont pas des entités isolées, ils dépendent du contexte formé des classes pour leur existence et leur exécution.
- L'implémentation des aspects peut être étroitement couplée à leur contexte tissé et dans ce cas, ils dépendent de la représentation interne et la mise en œuvre des classes dans lesquelles ils ont été tissés. Les modifications apportées à ces classes se propagent aux aspects.

- Les dépendances de contrôle et des données ne sont pas facilement apparentes à partir du code source des aspects ou des classes. En effet, en raison de la nature du processus de tissage, le développeur de classes ou des aspects ne connaît ni le flux de contrôle, ni la structure des flux de données résultant de l'artefact tissé. Ainsi, la détection des échecs correspondants aux défauts peut être difficile.
- Les comportements émergents. L'interaction entre classes et aspects d'un coté et entre aspects et aspects de l'autre peut être une source d'un comportement complexe dit *émergent*. Si une faute est détectée, il sera difficile de localiser son origine. En effet, la cause d'une faute peut résider dans la mise en œuvre d'une classe ou d'un aspect, ou encore peut être un effet de bord lié à un ordre de tissage particulier d'un ensemble d'aspects.

Face à ces caractéristiques les activités de test s'annoncent difficiles à mettre en œuvre. Dans la suite, nous présentons une analyse de la signification du test, en présence des caractéristiques citées ci-dessus et nous décrivons la nature des fautes possibles.

2 Test des aspects et modèles de défauts

L'entité de l'aspect n'est pas semblable à la classe. Elle n'est pas instantiable et son contexte est lié étroitement au contexte du tissage. Ce fait conduit à changer les dépendances de données et de contrôle du programme de base et en plus à en ajouter de nouvelles. Les défauts peuvent, alors, apparaître soit de l'implémentation d'une classe ou d'un aspect, ou de l'interaction des deux. Par conséquent, les approches de test de la programmation orientée-objets ne sont plus valides et on se retrouve contraints de chercher de nouvelles approches adéquates au paradigme orienté-aspects.

En cas de défaillance, le premier défi est de diagnostiquer l'échec et de détecter le défaut. Pour les programmes non orienté-aspects, il suffit d'examiner le code et peut-être les instructions pour isoler et localiser le défaut. Les échecs dans un programme orienté-aspects suivent une approche similaire. Cependant pour détecter un défaut c'est le code avec le tissage d'aspect qui doit être examiné.

R.Alixander et J.Bieman [1] identifient 4 sources de défauts dans les programmes orientés-aspects. Nous les citons ci-après.

- Le défaut réside dans une partie du code de base qui n'est pas affectée par le processus de tissage. Le défaut propre au code de base, pourrait se produire s'il n'y avait pas de tissage.

- Le défaut réside dans une partie spécifique du code de l'aspect qui est isolée du contexte de tissage. le défaut serait alors présent dans toute composition qui comprend l'aspect. La faute réside dans le code de l'aspect qui est indépendant des dépendances des données et de contrôle induites par le processus de tissage.
- Le défaut est une propriété émergente créée par l'interaction entre un aspect et le code de base. Ce défaut se produit lorsque le tissage introduit de nouvelles dépendances de données ou de contrôle qui n'étaient pas présentes dans le code de base ou le code d'un seul aspect. Ces nouvelles dépendances résultent de l'intégration et l'interaction du code et de données entre la le code de base et l'aspect.
- Le défaut est une propriété émergente créée par l'interaction entre plusieurs aspects tissés au code de base. La difficulté de localiser la source de défauts est aggravée lorsque plusieurs aspects sont en cause. Le défaut peut survenir uniquement avec une permutation des aspects lors du tissage avec le code de base.

Parallèlement à l'identification de ces sources de défauts R. Alexander et J. Bieman proposent six modèles de défauts [1].

a- Modèle 1 : Désignation incorrecte du point de coupure. La conception d'un point de coupure consiste à déterminer l'ensemble des points de jointure qui sont interceptés dans le programme principal pour l'exécution des advices associées. La désignation incorrecte de cet ensemble peut conduire à des comportements non désirés du programme final qui se répartissent en deux scénarii possibles.

1. L'ensemble de point de jointure est plus fort que nécessaire. Cas ou l'ensemble de point de jointure est plus fort par rapport à l'ensemble vraiment exigé par la préoccupation concernée. Dans ce cas, de nouveaux comportements sont ajoutés par erreur.
2. L'ensemble de point de jointure est plus faible que nécessaire. Cas ou l'ensemble de points de jointures est incomplet par rapport à l'ensemble vraiment nécessaire. Donc l'aspect échoue partiellement à appliquer ses advices ce qui engendre une incomplétude dans l'application des préoccupations.

b- Modèle 2 : Précédence incorrecte des aspects. Lorsque les ensembles des points de jointures définis aux points de coupure de plusieurs aspects d'un même programme sont en

interaction, l'ordre de précédence selon lequel les advices de ces aspects sont tissées (pour l'exécution) aux points de jointure affecte le comportement du système.

c- Modèle 3 : Echec dû au non respect des post-conditions implémentées dans le code de base. Les post-conditions, qui établissent des contrats entre les méthodes appelantes et les méthodes appelées, qui sont valides dans le code de base avant le tissage, doivent préserver leur validité après le tissage. Quand les advices produisent des effets qui violent la validité de certaines post-conditions, le comportement du système peut devenir incorrect.

d- Modèle 4 : Echec dû à la non préservation des états invariants. Les états invariants qui sont prouvés valides doivent être préservés après tissage pour garantir un comportement correct du système.

e- Modèle 5 : Mauvaise orientation du flux de contrôle. La désignation d'un point de coupure définit l'ensemble des points de jointures qui ne sont pas tous évalués statiquement. Les points de coupure primitifs évalués dynamiquement (tel que : *cflow*, *within*) peuvent être la source d'erreurs difficiles à localiser.

f- Modèle 6 : Changement incorrect dans les dépendances de contrôles. Le tissage des consignes « around » permet le remplacement du flux original du programme par du nouveau code défini au niveau de l'aspect. L'exécution originale peut être reprise mais la structure du flux de contrôle et les dépendances correspondantes sont changées. Cela peut modifier la sémantique des données ou des méthodes.

3 Les approches de test orientées-aspects existantes

Les techniques de test consistent à découvrir les différents types de défauts et de fournir les critères de couverture du code par la génération des cas de test. Pendant le test du programme orienté-aspects et après tissage, certain défauts peuvent avoir comme origine les concepts apportés par l'approche orientée-aspects, et d'autres les concepts du paradigme de programmation orientée-objets. En plus d'autre peuvent avoir comme source le tissage des concepts des deux paradigmes. Le processus de tissage complique donc le diagnostique et la localisation de l'origine.

La plupart des techniques de test orientées-aspects proposées actuellement par les chercheurs sont des extensions des approches de test orientées-objets et en peut les classées en techniques basées spécification (techniques fonctionnelles ou encore « boite noir ») et techniques basées implémentation (techniques structurelles ou encore « boite blanche »)

3.1 Techniques de test fonctionnel orienté-aspects

Ces techniques de test fonctionnelles consistent à vérifier si le programme satisfait ou non les besoins de spécification, en distingue deux types de techniques de test fonctionnelles : Les techniques de test formelles et les techniques de test semi-formelles. Nous les décrivons dans ce qui suit.

3.1.1 Les techniques de test formelles

Très peu de techniques formelles sont actuellement proposées pour le test des programmes orientés-aspects. La majorité étant inspirée des techniques déjà proposés pour le test formel des programmes orientés-objets. On distingue deux principales approches : la vérification basée-modèle et Vérification basée sur le modèle checking.

Vérification basée-modèle. G. Denaro & M. Manga ont été guidés dans leurs travaux par le problème de l'interblocage qui représente une propriété importante des systèmes informatiques [5]. L'implémentation de la concurrence des différentes préoccupations dans un programme orienté-aspects nécessite la dérivation des modèles adéquats pour l'application des techniques de vérification formelles. La technique formelle appropriée est celle appliquée pour démontrer la satisfaction des propriétés spécifiées ou pour donner un contre exemple avec des comportements non désirés. L'approche consiste en 6 étapes :

1. Sélectionner les propriétés générales d'une préoccupation donnée.
2. Sélectionner une technique de vérification formelle.
3. Sélectionner l'aspect qui encapsule le code approprié de la préoccupation concernée.
4. Utiliser le code de l'aspect pour dériver un modèle de vérification pour appliquer la technique formelle sélectionnée.
5. Spécifier les propriétés sélectionnées avec le formalisme approprié.
6. Créer un rapport de succès ou génération d'un contre exemple puis analyse.

Vérification basée sur le modèle checking. N. Ubayashi & T. Tamai [14] proposent une approche de vérification automatique utilisant le modèle checking. Ce dernier vérifie la validité de la concurrence d'un automate d'état fini qui peut représenter une conception LSI, des protocoles de communication ou la concurrence des systèmes logiciels. Il est utilisé pour vérifier les propriétés globale d'un programme orienté-aspects via une vérification automatique de leur spécification de besoins en utilisant la logique temporelle pour la description de ces derniers.

Le cadre du modelé checking proposé pour le test de la POA, est la composition de plusieurs aspects contrôleurs. Les propriétés qu'on souhaite tester sont exprimées sous forme d'un

aspect contrôleur qui décrit les prés et les post-conditions utilisant les points de jointures. L'aspect contrôleur est préparé pour correspondre à une propriété à tester donnée.

3.1.2 Les techniques de test semi-formelles

On distingue principalement deux approches : génération de test basé sur la modélisation et le test basé-état.

Génération de test basé sur la modélisation. W.Xu & D.xu [20] proposent une approche de vérification basée sur la modélisation pour la génération de cas de test des programmes orientés-aspects. Dans cette approche la modélisation consiste en des diagrammes de classes, des diagrammes d'aspects et des diagrammes de séquence. Ces derniers sont utilisés pour modéliser les méthodes des classes et les advices des aspects dans une seule entité. Le principe de l'approche consiste à exploiter le diagramme de séquence tissé pour construire le graphe de flux correspondant à un critère de couverture donné, et plus loin développer un graphe de l'arbre de flux, ou chaque chemin de l'arbre est un test. Comme extension du diagramme de classe UML, le diagramme de classe aspectuel (aspectual class diagram) représente une collection de déclarations statiques : classe, aspect, types, contenus, et relations. Le diagramme de séquence interprète les interactions entre les objets incluant les méthodes et les advices. Il capture le flux de messages ce qui facilite le tissage des aspects dans les classes. Le graphe de flux obtenu est transformé en arbre de flux, ou chaque chemin à partir d'un nœud d'une feuille à la racine représente une séquence de messages (requête) ou invocation de méthodes indiquant les cas de test.

Le test basé-état. D.Xu, W.xu, et K.N. Ygard [18] proposent une approche pour le test des programmes orientés-aspects qui est inspiré de celles appliquées pour les programmes orientés-objets en utilisant le graphe d'état « statecharts ». Pour tenir compte des caractéristiques spéciales de la POO tel que l'héritage. Le modèle d'état est adapté au FREE (Flattened Regular Expression). Les préoccupations qui entrecoupent le code de base (les éléments transversaux) modifient le flux de contrôle du programme de base. De ce fait, les relations de transitions d'états ne sont pas seulement changées mais encore des états supplémentaires sont introduits, ce qui conduit à une extension du modèle FREE pour supporter le modèle d'état aspectuel (ASM). Le ASM développé pour fournir un modèle testable au comportement de classes avec ajout des advices définies par les aspects qui sont additionnés dynamiquement quand un point de jointure spécifié est atteint. L'idée principale est de transformer le model d'état en arbre de transition d'états ou chaque chemin de la racine à un nœud d'une feuille représente un cas de test, cette transformation est effectuée via un algorithme qui supporte les comportements inattendus de l'aspect par l'application de plusieurs critères de couverture.

3.2 Techniques de test structurelles pour le test des programmes orientés–aspects

Les techniques de test structurelles souffrent souvent de la complexité due à l'intégration des nouveaux constructeurs de la POA et aussi des dépendances de contrôles aspect/aspect et aspect/code de base. La première technique de test structurel proposée spécialement pour le test des programmes orientés-aspects est celle de J. Zhao [21] datant de 2003. C'est une technique de test unitaire basée sur le flot de données pour le test des programmes orientés-aspects. D'autres techniques existent, nous les décrivons ci-après.

3.2.1 Test unitaire basé sur le flot de données

J.Zhao propose une technique de test unitaire basée sur le flot de données [21]. Cette technique consiste à tester comment les valeurs associées aux variables affectent le comportement d'un système. Elle permet de tester deux types d'unités pour un programme orienté-aspects (la classe et l'aspect). Pour chaque unité le test est exécuté aux 3 niveaux suivants :

a- Intra–module. Application pour des modules individuels tels que : advice, introduction, méthodes.

b- Inter-module. Application pour des modules publiques apportés par d'autre qu'ils appellent, pouvant être un aspect ou une classe.

c- Intra-aspect ou intra-classe. Application pour des modules qui peuvent être accédés en dehors d'un aspect ou d'une classe et qui peuvent être invoqués dans n'importe quel ordre.

Le test de flot de données fait principalement la mise au point de valeurs définies pour chaque variable du programme par l'exécution des sous chemins à partir de la définition de certains points dans le programme dans lequel la variable est utilisée.

Le graphe de flot de contrôle est construit pour calculer les couples (définition, utilisation) pour un aspect ou une classe et aussi pour guider la sélection des tests appropriés.

Les outils de développement qui prennent en entrée un programme aspectJ, l'analysent pour dériver les informations concernant le flot de données et de contrôle nécessaire pour la construction du graphe de flux de contrôle et calculent les couples (définition, utilisation) et génèrent les différents cas de test pour chaque module, groupe de modules, des aspects et des classes, Puis le générateur de cas de test utilise les couples (définition, utilisation) pour générer les différents cas de test pour chaque module, groupes de module, et pour chaque aspect, et chaque classe, qui sont ensuite utilisés par le « test driver » qui lit les cas de test, contrôle leur syntaxe, et puis les exécute. Finalement un contrôle la correction des résultats est effectué.

Remarque

La technique utilise les graphes de flux de contrôle pour calculer les couples (définition, utilisation) pour les aspects et les classes moyennant un algorithme spécifique.

3.2.2 Le graphe de flux d'aspect

Celle technique a été proposée par Weifug Xu et al, [16]. Elle utilise une hybridation des modèles de test qui combine un modèle de test basé-responsabilité et un modèles de test basé-implémentation. Un graphe dit AFG (aspect flow graph) est construit par la combinaison de ASSM (Aspect Scope State Model) et le graphe de flux des méthodes et des advices. Le ASSM est un diagramme de transition d'état pour un programme orienté-aspects qui est créé à partir des modèles d'états :

1- FREE un modèle d'état qui est un diagramme de transition d'état,

2- Le modèle d'état d'aspect qui comprend quatre états et trois arcs.

Le graphe de flux des méthodes et des advices représente le flux de contrôle à l'intérieur des méthodes et des advices. La contribution principale est de créer des suites de test traitables par le code de base.

3.2.3 L'intégration de test unitaire à travers IACFG (Inter-procedural Aspect Control Flow Graph)

Bernardi et Lucca [4] montrent que les programmes orientés-aspects peuvent être testés par l'estimation de l'interaction inter-procédurale entre les advices et les méthodes. Les auteurs fondent leur travail sur le graphe de flux de contrôle inter-procédural d'aspect (Inter-procedural Aspect Control Flow Graph), due à l'ordre d'interaction entre les différents constructeurs de la POA et celui de la POO. Plusieurs types de défauts peuvent exister et il est dur de comprendre leurs origines. Si ces constructeurs sont utilisés encore systématiquement alors il est très difficile de comprendre l'origine des défauts. Il est nécessaire d'avoir l'information complète étape par étape du processus d'interaction des aspects et le code de base. Bernardi et Lucca affirment que les caractéristiques comme « la modification structurelle », « l'introduction des champs », « les advices autours » n'ont pas été considérés précédemment. Maintenant ces caractéristiques sont exploitées à travers le modèle du test qu'ils proposent. Ces derniers mettent spécialement en considération les changements suivants :

- Le premier est « Inter-procedural Control Flow Based Alteration », qui est le changement du flux de contrôle du code des classes de base, due à des intégrations à certains points de jointures.

- Le deuxième est « System's structure transformation » qui est en fait la modification de la structure d'une classe. ou d'une interface après l'intégration des constructeurs spéciaux de la POA.

Les auteurs soulignent dans leur stratégie du test, les défauts qui sont reliés à l'expression des points de coupures et des advices. Pour l'implémentation de la technique les auteurs introduisent 6 critères à suivre.

3.2.4 Tester les programmes orientés-aspects comme les programmes orientés-objets

J.Zhao et R.Alexander [22] s'intéressent à la faisabilité de tester un programme AspectJ comme programme Java. Dans cette approche Zhao et Alexander proposent l'idée de décompiler le code tissé puis le tester comme étant un programme orienté-objets. Les auteurs partent du fait que la décompilation des bytecodes tissés d'un programme orienté-aspects produit un code OO qui est similaire au code OO original avec un certain nombre de modifications. Il n'est pas, alors, nécessaire de développer de nouvelles techniques de test spécifiques pour les nouveaux constructeurs introduits par la POA car les techniques OO sont satisfaisantes. L'approche consiste en :

1- le tisseur AspectJ utilise une composition statique à la compilation (cas de Eclipse avec AJDT) pour développer, compiler et obtenir les bytecodes tissés.

2- Après l'obtention des bytecodes tissés, utiliser JODE (un décompilateur java) pour décompiler les bytecodes tissés.

3- Le code java décompilé n'est rien qu'un programme java normal qu'on exécute via les cas de test avec vérification des résultats.

3.2.5 Test basé défauts

La technique du test basé défauts a été proposée par J. Zhao & R. Alexander [22] en 2007. La technique préconise un processus de test basé sur les différents types de défauts. Pour une meilleure efficacité de l'utilisation des tests basés modèles, les auteurs expliquent les modèles de dépendances et les modèles d'interaction entre le code de base et le code orienté-aspects. La technique propose 3 types de modèles :

1- modèle de dépendance

2- modèle d'interaction

3- modèle de défauts

La figure suivante montre la structure complète de la technique

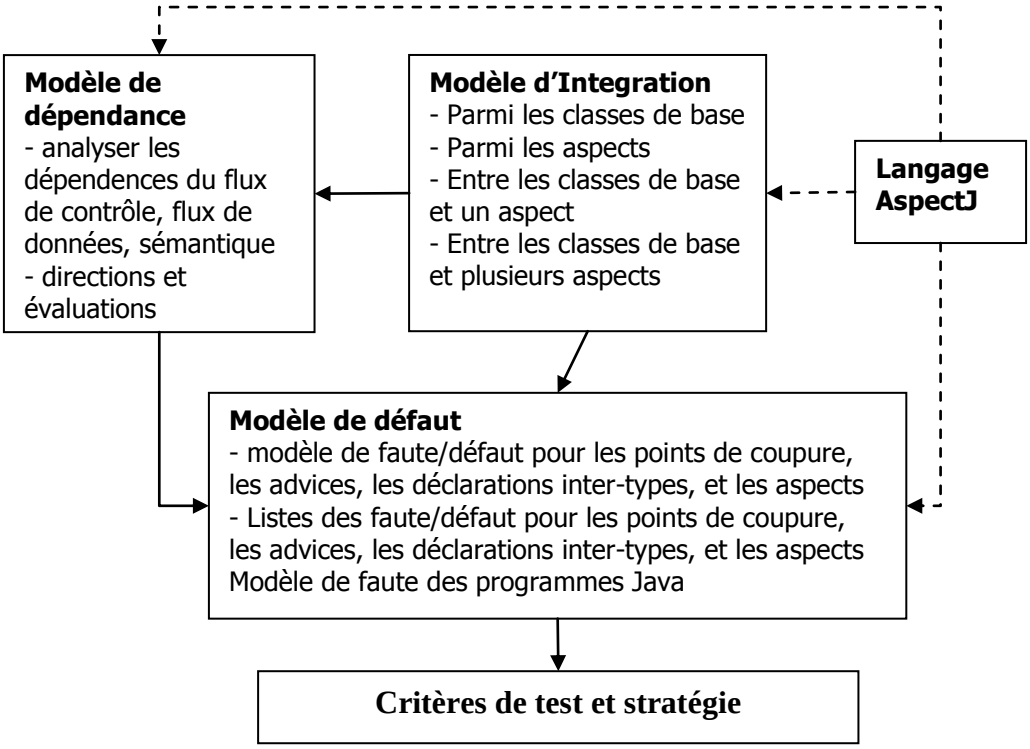


Figure 3.1 : Modèle d'architecture de la technique du test basé défauts

3.2.6 La génération des mutants pour le test mutationnel des points de coupure

Le test basé défauts est une approche ou la conception des tests de données est utilisée pour démontrer l'absence de l'ensemble des défauts pré-spécifiés (typiquement fréquents). Le test mutationnel est une technique de test utilisée pour injecter les défauts dans un programme existant. Anbalagan et Xie [2] tentent d'écrire les expressions de façon incorrecte. Cela cause un échec des aspects. Puis ils testent les points de coupures pour ces expressions.

Le test mutationnel des points de coupures inclut deux étapes :

Etape 1 : La création des mutants effectifs

Etape 2 : Le test de ces mutants par l'utilisation de la conception de test de données (designed test data)

Le nombre des mutants pour l'expression d'un point de coupure est toujours grand dû à l'utilisation des wildcards. Il est fatigant d'identifier manuellement les mutants effectifs. Anbalagan et Xie ont développé un framework qui génère automatiquement les mutants d'une expression d'un point de coupure et identifie les mutants qui ressemblent étroitement au expressions originales, puis ils utilisent le test de données pour les classes tissées contre ces mutants de l'exécution de test mutationnel.

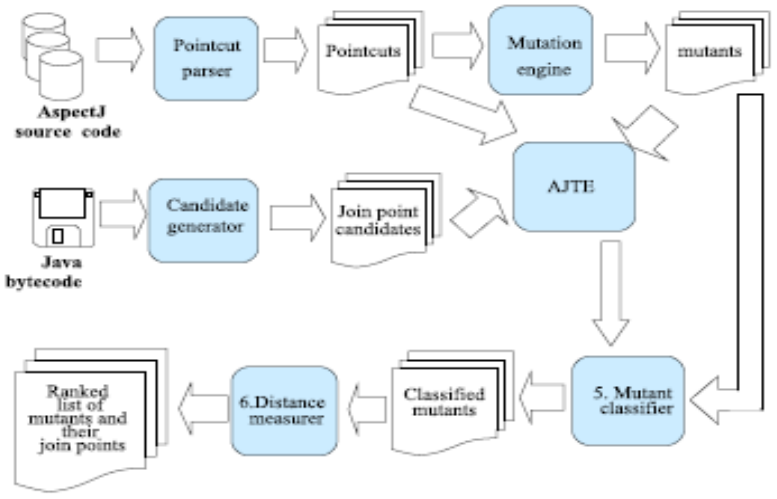


Figure 3.2 : Le framework du test des mutants

3.3 Techniques de qualité

3.3.1 La sélection de test de régression

J Zhao, T. Xie et N. Li [24] ont proposé leur technique en 2006 partant du travail de Harrold et al. dans la sécurité de la technique de sélection de test de régression pour les programmes Java. Les auteurs introduisent le concept de « Dangerous arcs » qui est la variation entres les arcs original et les arcs modifiés dans le graphe de flux de contrôle. Ces arcs représentent le code de base et le code tissé respectivement, les auteurs utilisent le GFC (graphe du flux de contrôle) pour la modélisation des différents composants des programmes AspectJ. Ils donnent la solution pour la modélisation individuelle des modules comme les advices, les méthodes, les méthodes inter-type. Initialement ils commencent par la modélisation d'aspects individuels puis les interactions aspect-classe et enfin ils modélisent le programme en totalité pour représenter les relations comportementales statiques et dynamiques des aspects et des classes.

Pour la modélisation de ces relations les auteurs exploitent le GFC et développent le GFCA (graphe du flux de contrôle d'aspects) pour la modélisation individuelle d'aspect et GFCS (graphe de flux de contrôle du système) pour la modélisation complète d'un programme AspectJ. La modélisation des différentes versions d'un programme AspectJ aide à la détection des arcs dangereux, qui finalement aide à la sélection des cas de test effectifs.

3.3.2 La technique de sélection des tests de régression

Guoqing Xu [19] introduirent une technique de sélection de test sécurisée pour les tests de régression en 2006. Cette technique est applicable dans deux scénarii. Le premier suppose un logiciel orienté-objets P et il y a une version orientée-aspects de ce logiciel P'. Le second

suppose qu'il y a deux logiciels orientés-aspects, ou le premier est l'original A et le deuxième est la version modifiée du logiciel actuel A'.

Un nouveaux cadre (framework) pour le test des programmes orienté aspect appelé RETESA (Regression Test Selection for Aspect-Oriented programs) est introduit par Guoqing Xu [19]. Soit un programme orienté-aspects qui est une amélioration d'un programme orienté-objets. Il est évident que le graphe de flux de contrôle est différent pour les deux programmes ; cette technique, premièrement, sélectionne les cas de test à partir de l'ancienne suite de tests et puis note les différences dynamiquement et ré-sélectionne de nouveaux les cas de test pour la version améliorée du logiciel. Le terme « dangerous arc » est utilisé dans cette technique pour les autres effets de bords dans le GFC qui sont générés par le processus de tissage. Un algorithme de parcours de graphe est utilisé pour identifier ces effets à partir de GFC.

3.3.3 La détection de redondance dans la création des cas de test

Le framework Raspect est proposé par Tao Xie et Jianjun Zhao, Dark Marinov, et David Notkin [17], pour la détection de redondances dans le test unitaire des programmes AspectJ. Ils ont introduit 3 niveaux d'unités d'un programme AspectJ : advised methods, *advice*, inter-type *methods*, et montrent comment détecter à chaque niveau les tests redondants qui ne produisent pas de nouveaux comportements. L'approche sélectionne seulement les tests non redondants à partir de la suite de tests générés automatiquement. Cela permet au développeur de passer moins de temps dans l'inspection de cet ensemble réduit de tests. Les auteurs implémentent Raspect et l'appliquent dans 12 sujets pris à partir d'une variété de sources. L'expérience a montré que Raspect peut effectivement réduire le nombre de tests générés pour l'inspection d'un programme AspectJ.

Raspect exécute les étapes suivantes :

1- Compiler et tisser les aspects et les classes de base dans une classe bytecode utilisant le compilateur AspectJ.

2- Génère les tests unitaires pour les classes tissées en utilisant les outils de génération de test basés sur la classe bytecode exemple Jtest (pour certains types d'advice, il faut utiliser le framework Aspectra).

3- Compiler et tisser les aspects et générer le test des classes bytecode utilisant le compilateur d'AspectJ.

4- Détecter et enlever les tests redondants pour les 3 types d'unités.

- Pour chaque advised method, traiter la classe tissée comme une classe sous test et l'advised method elle même comme une méthode sous test.

- Pour chaque advice, traiter la classe aspect comme une classe sous test et l'advice comme une méthode sous test.
- Pour chaque méthode inter-type, traiter l'aspect comme une classe sous test et la méthode inter-type comme une méthode sous test.

L'utilisation de Raspect pour la détection des tests redondants pour les advised methods, les advices, et les méthodes inter-type suppose que les méthodes sont déterministes : Pour chaque méthode, deux exécutions qui débutent avec le même état (atteignable à partir du récepteur et les arguments de méthodes) ont le même comportement. En particulier ça signifie que Raspect ne traite pas les cas de multi-threading.

3.4 Les Frameworks

3.4.1 Aspectra

Aspectra [17] est un cadre de génération des tests d'entrés (inputs) pour le test du comportement aspectuel. Aspectra classe l'exécution en 4 types de méthodes dans un programme AspectJ : advised methods définies dans les classes de base, advices, intertype methods, et public non-advice methods définies dans les classes aspect. Les développeurs peuvent construire les classes de bases et tisser les aspects dans ces dernières pour produire les classes tissées, qui peuvent être insérées dans les outils de génération de test. Pour activer les méthodes définies dans les aspects pendant le test de génération, Aspectra développe un mécanisme de couverture pour préparer les classes tissées à être testées par les outils de génération de test existant. Aspectra évalue la génération de test des programmes java pour générer les tests d'entrée pour les programmes AspectJ. Comme les tests de génération initiales peuvent être insuffisants et ne couvrent pas le comportement aspectuel d'aspect, Aspectra définie et mesure la couverture des branches aspectuelles et l'interaction de couverture pour guider les développeurs et améliorer les constructeurs des classes de bases et la génération des tests.

Aspectra fait automatiquement la synthèse des classes de couvertures pour chaque constructeur des classes de base, pour les aspects, puis pour les classes de couverture. Aspectra est inséré aux outils de génération de test existants.

3.4.2 Wrasp

Wrasp est un cadre pour l'intégration des tests [17], il se compose de deux parties, la première fait la synthèse des classes de couverture (wrapper class) pour les classes de base, la deuxième partie est adaptée aux techniques de génération de test existantes par le traitement des classes de couverture comme des classes sous test.

La première partie consiste en plusieurs outils de génération de test automatique et est basée sur les bytecodes Java. Il y a 5 étapes pour la génération des tests d'intégration basés sur la classe de couverture.

1- Compiler et tisser les classes de base et les aspects dans une classe bytecode en utilisant le compilateur AspectJ.

2- Faire la synthèse des classes de couverture pour les classes de base en considérant la classe bytecode tissée.

3- Compiler et tisser les classes de base, les classes de couverture et les aspects dans une classe bytecode utilisant le compilateur AspectJ.

4- Générer les tests pour les classes de couverture tissées en utilisant les outils de génération de test existants

5- Compiler les classes de test générées en une classe bytecode en utilisant le compilateur Java.

La deuxième partie fait la synthèse des classes de couvertures pour les classes de base sous test. Dans cette classe de couverture, on trouve la synthèse des méthodes de couvertures pour chaque méthode publique dans les classes de bases. Les méthodes de couverture invoquent les méthodes de couvertures dans les classes de bases.

Le framework Wrasp pour la génération de test unitaire produit le test input pour traiter directement un advice dans un aspect de façon isolée. Wrasp traite les aspects comme des classes sous test et les advices des aspects comme des méthodes des classes sous test.

Conclusion

Dans ce chapitre nous avons présenté un panorama des approches de test orientées-aspects en les classant en techniques structurelles, techniques fonctionnelles, techniques de qualité, et enfin les cadres (framework).

Le but de chacune des approches est de tester complètement et efficacement un programme orienté-aspects. Peut-on considérer que ce but est atteint ? Est-ce-que le test orienté-aspect ne souffre plus d'aucun problème ?

Nous répondrons à ces questions dans le chapitre suivant où nous proposons notre propre approche de test.

CHAPITRE 4
TECHNIQUE STRUCTURELLE BASÉE SUR LES POINTS DE JOINTURE DYNAMIQUES POUR LE TEST DE LA POA

Comme vu dans le chapitre précédent, il existe actuellement des tentatives qui proposent des approches de test qui tiennent compte des préoccupations, la plupart étant inspirées de ce qui est appliqué aux programmes orientés-objets, et souffrent d'insuffisances qui les rendent incapables d'être des références solides pour le test des programmes orientés-aspects.

Les points qui intéressent les aspects dans le code de base, sont les points de jointures. Une approche qui traite les cas de test liés à ces derniers doit tenir compte en particulier des points de jointure qui sont dynamiques. Ces derniers provoquent les défauts liés à la structure du programme au moment de l'exécution.

Pour cela dans ce chapitre nous proposons une technique de test structurelle qui tente de combler les insuffisances des approches de test actuelles, par une évaluation statique complète de la structure du programme couvrant tous les cas dynamiques possibles. Nous réalisons cet objectif via un algorithme que nous avons nommé GCC (Graphe de Contrôle Complet) qui traite les trois points suivants :

1. Tissage des aspects au graphe de flux de contrôle du code de base.
2. Évaluation complète des primitives dynamiques.
3. Définition de l'ordre de tissage des aspects.

Dans ce chapitre, nous commençons par une comparaison des approches de test orientées-aspects qui met en évidence leurs insuffisances puis nous décrivons de manière détaillée la technique que nous proposons.

1 Comparaison des approches de test

Une analyse approfondie et une étude comparative des approches de test présentées dans le chapitre précédent est nécessaire pour évaluer la capacité des approches existantes à tester les programmes orientés-aspects. En effet, ce paradigme a introduit de nouveaux constructeurs, de nouvelles dépendances et de nouveaux mécanismes rendant les approches de test de ces paradigmes incapables de tester convenablement les programmes orientés-aspects. La comparaison que nous faisons ci-après se base sur deux facettes. La première est celle des avantages que procure chaque approche et ses insuffisances et la seconde est celle de la détection des modèles de défauts.

1.1 Comparaison selon les caractéristiques des approches

On distingue deux classes de méthodes : les méthodes structurelles s'appuyant sur l'analyse du code source de l'application dans lequel la structure interne doit être accessible, et les méthodes fonctionnelles qui s'intéresse aux besoins fonctionnels du système et où la connaissance de la structure interne n'est pas nécessaire. Pour une étude comparatives il est intéressant de comparés chaque classe de méthodes de test à part, la comparaison se base sur les caractéristiques de chaque approche (i.e. avantages et inconvénients).

1.1.1 Les approches de test fonctionnelles

Approche	Avantages	Insuffisances
Vérification basée-modèle	Vérifie les propriétés globales du système (comme l'inter blocage) et garantir leur préservation / absence.	1- Analyse insuffisante des interactions entre les classes et les aspects. 2-Difficulté à localiser l'erreur.
Vérification basée sur le model checking	Pertinent pour le test de la POA car elle vérifie les propriétés globales du système.	1- Les aspects et les classes doivent être testés séparément car les défauts concernent les propriétés globales capable d'être cachés par des unités individuelles (classes ou aspects). 2- Le cadre du model checking est incapable de couvrir toutes les propriétés du système comme le problème de l'interblocage.
Génération de cas de test basé sur la modélisation	Extension d'UML par un diagramme de classe aspectuel, le diagramme de séquence capture le flux des messages et facilite le tissage d'aspects dans les classes. Le diagramme de séquence après tissage est transformé en arbre de flux. Chaque chemin à partir d'un nœud d'une feuille à la racine indique un cas de test.	1- Génération des cas de test non conditionnés (les tests que nous avons besoin de générer pour tester les préoccupations transversales). 2- Insuffisance de la description des détails d'implémentation. 3- Certains cas peuvent être oubliés à cause des points de coupure contenant des primitives dynamiques (ex cflow, within, …) qui définissant l'ensemble des points de jointure qui sont difficiles à évaluer statiquement.
Test basé–état	Permet de révéler les défauts d'une spécification incorrecte d'un point de coupure et les échecs dus à la non préservation des états invariants	1- Explosion combinatoire des états 2- Insuffisance de la description des détails d'implémentation 3- Difficulté à analyser les chemins intra-classe, si cette dernière décrit seulement les méthodes dans les interfaces de haut niveau.

Tableau 4.1 : Comparaison des approches de test fonctionnelles

1.1.2 Les approches de test structurelles

Approche	Avantages	Insuffisances
Test unitaire basé sur le flot de données	Explication détaillée du comportement des advices et des méthodes après tissage. Capacité à détecter les échecs dus à la préservation des états invariants parce que leur défauts sont associés aux erreurs du flot de données.	Les advices « around » peuvent affectées le comportement de contrôle ou les dépendances de données, qui touchent directement les couples (définition, utilisation) des méthodes.
Tester les POA comme les POO	Les bytecodes décompilés ne sont rien d'autre que des programmes Java (orienté objet), qui peuvent être testés en utilisant les techniques de test orientées objets normales.	- Cette technique est pratique mais uniquement pour des applications simples. - Difficulté à localiser les défauts dans le programme AspectJ source. - Les aspects et les classes devraient être testés séparément pour préserver les avantages de séparation des préoccupations.
Test basé défauts pour les Programmes aspectJ	Technique efficace parce que son utilisation révèle la présence ou l'absence de défauts spécifiques dans un logiciel.	Absence d'une méthodologie du test.
Le graphe de flux d'aspect pour le test des POA	- Hybridation du modèle de test basé responsabilité et du modèle de test basé implémentation. - Capable de manipuler les advices appliquées dynamiquement. - Capable de détecter les défauts liés au changement dans les dépendances de contrôle (modèle de défaut 6).	- Les advices appliquées dynamiquement, dépendent de chaque contexte d'exécution. - Le graphe de flux d'aspect est un modèle pour le tissage statique. Alors il est incapable de détecter les fautes liés à la mauvaise orientation du flux de contrôle (modèle de défaut 5)
Le test des mutants des points de coupures dans les POA	- La génération des mutants importants et la détection des mutants équivalents pour une efficacité du test mutationnel. - La technique peut effectivement génère des mutants qui peuvent être en suite testé par l'utilisation des données des tests déjà conçus.	- Explosion combinatoire dans la génération des mutants. - Le cout du test est relativement élevé.

Tableau 4.2 : Comparaison des approches de test structurelles

A partir de ce tableau comparatif nous concluons que chacune de ces approches possède de bons avantages mais souffrent aussi d'insuffisances. Ces dernières rendent ces approches incomplètes et incapables de détecter tous les types de défauts qui peuvent exister dans un programme orienté-aspects. Dans la suite, nous comparons les approches en fonctions des modèles de défauts.

1.2 Comparaison selon les modèles de défauts

L'utilisation du test permet la révélation de la présence ou l'absence des défauts dans un logiciel. Les modèles de défauts ont été présentés dans le chapitre précédent, nous les rappelant dans ce qui suit.

1- Désignation incorrecte du point de coupure.
2- Précédence incorrecte des aspects.
3- Echec lié aux post-conditions.
4- Echec dû à la non préservation des états invariants.
5- Mauvaise orientation du flux de contrôle.
6- Changement incorrect dans les dépendances de contrôle.

Le résultat de notre comparaison est donnée dans le tableau suivant :

Les modèles de défauts / **Les approches**	**Capable de détecter**	**Incapable de détecter**
Vérification basée sur le model checking	6	1, 2, 3, 4, 5
Test basé–état	1, 4	2, 3, 5, 6
Test unitaire basé sur le flux de données	4	1, 2, 3, 5, 6
Le graphe de flux d'aspect pour le test des programmes orienté- aspect	6	1, 2, 3, 4, 5
Tester les programmes OA comme les programmes OO		1, 2, 3, 4, 5, 6
Test basé défauts pour les programmes AspectJ	3,6	1, 2, 4, 5
Génération des mutants pour le test mutationel des points de coupures dans un POA	1	2, 3, 4, 5, 6

Tableau 4.3 : Comparaison des approches de test OA selon les modèles de défauts.

1.3 Discussion

Les approches de test orientés-aspects existantes ont des avantages mais aussi soufrent d'insuffisances qui les rendent incapables de tester efficacement un programme orienté-aspects en couvrant tous les défauts possibles. Ces insuffisances varient d'une approche à l'autre selon le type de celle-ci (fonctionnel ou structurel). Même lorsqu'on considère les approches de même types ont remarque des différences qui viennent cette fois d'autres facteurs : tel que l'outil utilisé, leur fondement sur un existant ou pas, etc. Après ce que nous avons présenté dans ce chapitre nous pouvons classer les insuffisances des approches en trois catégories que nous présentons dans ce qui suit.

1.3.1 Les insuffisances des techniques basées-spécification

Les techniques de test basé spécification sont toutes inspirées des techniques déjà appliquées pour le test des programmes OO. A partir de la définition du test fonctionnel, ce type de test vérifie le comportement d'un logiciel par rapport à sa spécification, il ne s'intéresse pas à la structure du programme, tandis que le paradigme de programmation orienté-aspects ajoute de nouveaux constructeurs au paradigme OO qui touchent la structure interne du programme (les advices around, les déclarations inter-types, …). Cela rend ces techniques de test fonctionnelles moins adaptées et incomplètes pour le test des programmes. En particulier, il est difficile de localiser et de diagnostiquer la cause des défauts.

1.3.2 Les insuffisances des techniques basées-implémentation

Les techniques de test basées-implémentation sont motivées par des exemples d'applications simples. L'application de ces techniques sur des exemples simples peut ne pas présenter correctement la capacité de celle-ci. Certaines ne couvrent aucun des modèles de défauts présentés par R. Alexander et J. Beiman [1].

1.3.3 Les insuffisances communes

a- Fondement sur les techniques orienté- objet : La plus part des approches proposées pour le test des programmes orientés-aspects sont inspirés de celles déjà appliquées aux programmes orientés-objets. Ce fondement est insuffisant à cause des nouvelles dépendances aspect-classe et aspect-aspect. Ces dernières introduisent de nouvelles situations et compliquent le processus de test et la détection des défauts qui sont liés aux interactions aspect-classe et à la composition des aspects.

b- Non respect du principe de la séparation des préoccupations : Le test ne profite pas du principe clé qu'est la séparation des préoccupations comme c'est le cas des autres activités du développement (analyse, conception, implémentation).

c- La capacité a détecté les modèles de défauts : A partir de la comparaison selon les modèles de défauts, on trouve que chacune des approches est capable de détecter un certain nombre de défauts parmi les six modèles de défauts proposés par R.Alexander et J.Biemman [1] ainsi que d'autres défauts. Aucune des approches n'est capable de détecter les 6 modèles. En particulier le modelé 2 (la précédence incorrecte des aspects) et le modèle 5 (l'évaluation des primitives dynamiques) qui sont étroitement liés aux caractéristiques de l'entité d'aspect et au contexte d'exécution du programme orienté aspect sous test.

2 Proposition d'une nouvelle technique structurelle basée sur les points de jointure dynamiques

L'étude comparative précédente nous a permis de faire le point sur les approches de test actuelles en situant leurs avantages et leurs inconvénients et leurs capacités à détecter les modèles de défauts. Nous constatons que l'approche orientée-aspects introduit un bouleversement important au niveau des approches de test et on voit apparaître une multitude d'approches différentes dont il convient d'opérer une unification. Partant de ce point, nous proposons une nouvelle approche de test des programmes orientés-aspects que nous baptisons « *technique structurelle basée sur les points de jointure dynamiques* ». Nous présentons notre approche dans ce qui suit.

2.1 Principe de la technique

Le paradigme de programmation orienté-aspects introduit de nouveaux constructeurs au paradigme orienté-objet qui touchent directement la structure interne du programme. Cette structure joue un rôle important dans la conception d'une technique de test orientée-aspects complète, où la majorité des modèles de défauts est en relation avec la structure interne du programme. Notre technique est basée principalement sur les modèles de défauts présentés par R.Alexander et J.Bieman dans [1].

Avant d'aller plus loin dans la description du principe de l'approche, il est nécessaire d'analyser les modèles de défauts.

2.1.1 Analyser des modèles de défauts

L'analyse consiste à attribuer à chacun des modèles de défauts les caractéristiques les plus pertinentes des approches de test proposées pour le test des programmes orientés-objets et aussi mettre en évidence clairement les modèles où les approches orientées-objets sont défaillantes. Cette analyse est illustrée dans le tableau ci-dessous.

	Les modèles de défaut	**Caractéristiques de l'approche de test pertinente**
01	Désignation incorrecte du point de coupure.	Basée sur le flux de contrôle
	L'ensemble de point de jointure est plus fort que le nécessaire	Les cas de test doivent traverser les branches incorrectes additionnées par erreur suite à l'interception de points de jointures correspondants
	L'ensemble de point de jointure est plus faible que le nécessaire	Si des branches sont manquantes a partir des points d'exécution des advices, les cas de test traditionnels peuvent exposés le défaut. Il s'agit de vérifier si le comportement sous test est altéré par l'exécution des advices manquants
02	Précédente incorrecte des aspects	Les approches de test OO sont incapable de détecter se genre de défauts
03	Echec dû à la non-vérification des post-conditions	Les cas de test qui expose les sorties (valeurs des variables) dépendants des post-conditions ou des états invariants sont capables de détecter ce genre de défauts
04	Echec dû à la préservation des états invariants.	
05	Mauvaise orientation du flux de contrôle	Les approches de test OO sont incapables de détecter ce genre de défauts
06	Changement incorrect dans les dépendances de contrôles	Utiliser un critère de couverture de branches qui traite les advices comme étant des appels aux méthodes paramétrés associés avec des changements normaux et exceptionnels du flux de contrôle

Tableau 4.4 : Analyse des modèles de défauts

a. Les modèles de défauts 1 et 6. Ils peuvent être détectés par une approche de test structurelle basée sur le flux de contrôle dont le critère de couverture est *tous les chemins* utilisés pour le test d'un programme orienté-objets, car elle est capable de détecter ces modèles de défauts avec une certaine spécialisation.

b. Les modèles de défauts 3 et 4. Ils peuvent être détectés par une approche de test structurelle basée sur le flot de données dont le critère de couverture est *tous les chemins*, et en étudiant les valeurs associées aux variables utilisées pour le test des programmes orientés-objets. Pour toute violation on associe les changements des variables dans des couples (définition, utilisation).

c. Les modelés de défauts 2 et 5. Ils sont des nouveaux types de défauts (par rapport à l'OO). Ils sont liés étroitement aux caractéristiques d'aspects et due aux dépendances aspect-aspect (le modèle de défaut 2), et aussi liés au contexte d'exécution (modèle de défaut 5) .

L'analyse de ces modèles de défauts nous permet de conclure, que les défauts qui affectent les programmes orientés-aspects sont de deux classes. Les défauts statiques détectables à la compilation (ou au tissage) (les défauts 1, 3, 4, 6) et les défauts dynamiques nécessitant une évaluation au moment de l'exécution (les défauts 2,5). Les approches de test OA existantes ont la capacité de détecter certains des défauts statiques.

En ce qui concerne les modèles de défauts 1et 6 beaucoup de techniques de test orientées-aspects qui sont proposées actuellement sont capables de les détecter. Parmi ces approches nous avons le *graphe de flux d'aspect pour le test des programmes orientés-aspects.* Celle technique est proposée par Weifug Xu et al [16], elle utilise une hybridation des modèles de test qui combine un modèle de test basé-responsabilité et un modèle de test basé-implémentation. AFG (Aspect Flow Graph) est construit par la combinaison d'ASSM (Aspect Scope State Model) et le graphe de flux des méthodes et des advices. L'ASSM est un diagramme de transition d'état pour un POA et il est créé à partir des modèles d'états

1- FREE un modèle d'état qui est un diagramme de transition d'état,

2- Le modèle d'état d'aspect qui comprend quatre états et 3 arêtes. Le graphe de flux des méthodes et des consignes représente le flux de contrôle à l'intérieur des méthodes et des consignes.

La contribution principale est de créer des suites de test traitables lors de l'exécution du code de base. Cette technique est capable de détecter le modèle de défaut 6 « Changement incorrect dans les dépendances de contrôles »

En ce qui concerne les modèles de défauts 3 et 4 des approches de test ont été proposées en se basant sur les techniques de test structurelles orientées-objet et plus précisément basées sur la couverture du flot de données des couples (définition, utilisation). Parmi ces approches on trouve le *Test unitaire basé sur le flux de données* (Data Flow Based Unit Testing, J.Zhao [21]) qui consiste à tester comment les valeurs associées aux variables affectent le comportement du système. Elle teste deux types d'unités pour un programme orienté-aspects : la classe et l'aspect. Pour chaque unité le test est exécuter à trois niveaux intra-module, inter-module, intra-aspect ou intra-classe. La technique utilise les graphes de flux de contrôle pour calculer les couples (définition, utilisation) pour les aspects et les classes via un algorithme qui les construit (le graphe de flux de contrôle). Cette technique

est capable de détecter les défauts 3, 4 grâce à l'évaluation des valeurs associées aux variables utilisées.

Pour les modèles de défauts 2, et 5, ils sont négligés par les approches de test issues de celles orientés-objets. Il est alors nécessaire de chercher une technique qui traite les cas de test liés aux types de défauts 2 et 5.

2.1.2 Les problèmes par les insuffisances des approches existantes

Dans certain cas plusieurs aspects nécessitent d'être tissés au même point de jointure. Dans le cas où le tisseur est statique comme AspectJ, l'ordre de tissage des aspects à ce point peut être incorrect car il est définit de façon statique alors qu'il exige une évaluation dynamique du contexte d'exécution.

a. Les défauts liés aux points de jointures dynamiques

- La définition des points de jointures. Le code de base est un programme orienté-objets (ou procédural) composé d'un ensemble de classes simples. Les points du programme qui sont sensibles aux aspects sont des points dans de le code de base dans lesquels les aspects sont tissés. Ils représentent les localisations de l'introduction des aspects dans le code de base. Leurs définitions jouent un rôle important dans l'exactitude de la structure du programme ainsi que le fonctionnement de son comportement. En plus, la définition de l'ensemble de point de jointure dans certain cas est difficile voir impossibles jusqu'au moment de l'exécution (utilisation des wildcard). De même, dans certains cas, plusieurs aspects sont tissés aux mêmes points de jointure. Dans ces cas la définition de l'ensemble de points de jointure nécessite une évaluation dynamique au moment de l'exécution. Donc ces points ont peut les nommés « point de jointure dynamiques ».

- Les primitives dynamiques. Permettent également de généraliser des coupes via des opérateurs logiques. Il est possible de faire du filtrage grâce à des mots clés spécifiques. Ces mots-clés permettent de préciser les points de jointures selon l'appelant, l'appelé, le flot de contrôle ou encore les arguments des fonctions. Associé à des opérateurs logiques, ces mots clés permettent de définir de manière très précise les points de jointures.

Les points de jointure dynamiques sont les points dans le code de base qui sont sensibles au tissage d'aspects au moment de l'exécution.

Si un point de jointure appartient à un point de coupure primitif dynamique, l'évaluation statique de cet ensemble conduit à oublier certains points (modèle de défauts 5)

Si le point de jointure est utilisé par plus d'un aspect aux même temps, cela conduit généralement à une précédence incorrect des aspects si cette précédence est définie statiquement (modèle de défaut 2).

b. Le test direct de l'artéfact tissé est insuffisant

Tester directement de l'artéfact tissé est insuffisant, car après tissage nous obtenons une seule configuration qui est statique (cas du tisseur statique d'AspectJ). C'est-à-dire un seul ordre de précédence de tissage des aspects. Cela est insuffisant surtout que l'ordre défini dans la configuration de l'artéfact tissé est inadéquat car il est presque aléatoire (non conforme au comportement attendu).

3 Les trois axes de la technique

Notre technique consiste à traiter les cas de test liés aux points de jointures dynamiques. La définition de ces points est liée à la structure du programme ce qui nécessite une approche de test structurelle. La technique proposée consiste à générer les cas de test dynamiques de manière statique. Elle complète les approches de test structurelles des programmes orientés-aspects dans la découverte des modèles de défauts 2 et 5 (associées au point de jointure dynamiques). La technique est construite via un algorithme qui traite les 3 axes suivants

3.1 Axe 1 : Le tissage des tests d'aspects au graphe du flux de contrôle du code de base

Partant de la constatation que POA (Classes + Aspects) = POO +Tissage d'Aspects, le test d'un programme orienté-aspects revient au test du programme orienté-objets plus des tests du tissage des aspects. Le test d'un POO utilise une approche de test structurelle basée sur le graphe de flux de contrôle (tous les chemins).

Si le graphe de flux de contrôle est construit sur la structure du programme après tissage plusieurs cas de test peuvent être oubliés (les cas qui sont liés aux points de jointure dynamiques), car le graphe de flux de contrôle est construit sur une structure du programme qui possède une seule configuration de l'ordre de tissage des aspects et cet ordre peut ne pas répondre aux besoins de spécification et l'évaluation des primitives dynamiques peut être incomplète.

Solution : Création des cas de test par le tissage de test des aspects sur le graphe de flux de contrôle du code de base.

3.2 Axe 2 : L'évaluation complète des primitives dynamiques (avec *wildcards*)

La désignation d'un point de coupure peut ne pas être entièrement résolue au moment du tissage. Si elle dispose des primitives dynamiques, elle nécessite une évaluation des conditions au moment de l'exécution.

L'idée consiste à créer un ensemble statique de points de jointure correspondant à l'évaluation dynamique d'un point de coupure. Nous considérons le cas comme étant une instruction de boucle, avec une différence, dans le cas des instructions de boucle le nombre de combinaisons possibles est limiter par une condition logique bien définie, alors que pour les primitives dynamiques le nombre de combinaison est liées à toutes les exécutions possibles ce qui nous oblige à considérer toutes ces exécutions.

Solution : Créer un ensemble statique de points de jointure correspondant à l'évaluation dynamique d'un point de coupure primitif dynamique. Avec le remplacement des conditions dynamiques par la valeur *vraie*.

3.3 Axe 3 : Définition de l'ordre de tissage d'aspect

Quand l'ordre de tissage des aspects a des contraintes dominantes qui doivent être utilisées pour spécifier la précédence correcte des aspects, le problème de la précédence incorrecte des aspects ne se pose pas. Mais en absence de telles contraintes, des erreurs de composition peuvent se produire.

Solution : Créer toutes les configurations possibles de l'ordre du tissage des aspects qui sont liés au même point de jointure par le même type d'advice.

Par exemple en considérant trois aspects (p1,p2,p3), les configurations possibles de précedence sont au nombre de six (3 !) :

(p1→p2→p3, p1→p3→p2, p2→p1→p3, p2→p3→p1, p3→p1→p2, p3→p2→p1)

4 L'algorithme du graphe du flux de contrôle complet

Nous proposons un algorithme dit GCC qui traduit les trois axes ci-dessus. Les entrées de l'algorithme sont :

1- un programme orienté-aspects (code de base et aspects)

2- Le graphe du flux de contrôle du code de base (l'ensemble des classes)

3- Les aspects

Les sorties sont :

1- Un graphe de flux de contrôle complet

2- Les chemins liés à l'évaluation des primitives dynamiques et à la précédence incorrecte des aspects

L'algorithme de la construction du graphe du flux de contrôle complet est le suivant :

Début
Pour chaque aspect A **faire**
Pour chaque point de coupure pc de l'aspect A **faire**
Si pc ≠ primitive dynamique **alors**
Identifier les points de jointure dans le GFC associés à pc
Ajouter les arcs correspondants aux consignes tissées aux points de jointures
Sinon // Pour obtenir tous les points de jointures
Remplacer la condition de la primitive par Vrai
Identifier les points de jointure dans le GFC associés au pc
Supprimer et ajouter les arcs correspondants aux advices tissées .
Fin si
Fin pour
Fin pour
/* L'ensemble des points de jointures capturés dans la partie haute de l'algorithme, représente tous les points de jointures du programme appelé « PJ » */
Pour chaque pj de PJ **faire**
Si (pj se trouve n fois dans l'ensemble) **et** (n>1) **et** (les advices associées à pj sont de même type, avant, après, autour) **alors** Ajouter les arcs nécessaires
Fin si
Fin pour
Fin

5 Exemple d'utilisation

5.1 La détection du modèle de défaut 2

Pour bien illustrer le fonctionnement de notre technique, nous donnons dans cette section une application exemple. Cette application dispose de deux classes et deux aspects qui afficher des messages.

Exemple L'application communication

Les classes

```
Public class Communication {     // La classe communication
   Public static void print(string[] message) {
     System.out.println (message) ;
   }
}
Public class Principale {     //  La classe principale
   Public static void main(string[] args) {
     Communication.print ("hello !")
   }
}
```

Les aspects

```
aspect QuestionAspect {    // l'aspect question
  Pointcut printcall1() :  call(void Communication.print(..))
  After printcall1(){
    System.out.println ("how are you, ");
  }
}
Public aspect LearnAspect {    // l'aspect learnaspect
  Pointcut printcall2() :  call(void Communication.print(..))
  After  printcall2() {
    System.out.println ("want to learn aspectJ? ");
  }
}
```

5.1.1 Les exécutions possibles

Dans cette application on a la situation ou plusieurs aspects doivent être tissés au même point de jointure qui est *Communication.print()*.

Dans ce cas la il y a deux résultats d'exécution possibles :

a. La première exécution. Si on précède à l'exécution de l'aspect *QuestionAspect* avant *LearnAspect*, nous obtenons comme résultat le message suivant

Hello ! how are you, want to learn aspectJ?

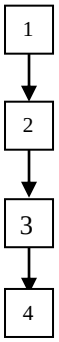
b. La deuxième exécution. Mais si on précède à l'exécution de l'aspect *QuestionAspect* avant *LearnAspect*, nous obtenons comme résultat le message suivant :

Hello ! want to learn aspectJ? how are you,

Les deux résultats sont différents, dans le premier cas le résultat est ce qu'on souhaite obtenir. Pour le deuxième cas le résultat est non souhaité.

5.1.2 Application de la technique sur un exemple

a- Construction du graphe de contrôle du code de base. Le code de base de notre exemple est composé des deux classes : la classe communication, et la classe principale. Leur graphe de contrôle est le suivant.

Ordre d'exécution des instructions	**Le graphe de contrôle**
1- début 2- la méthode main 3- la méthode communication.print() 4- fin.	

b- Application de l'algorithme GCC

```
1- Pour chaque pj de PJ faire

2- Si (pj se trouve n fois dans l'ensemble) et (n>1) et (les
advices associées à pj de même type, avant, après, autour)
alors

     2.1- Crée toute les configurations possibles de la
     précédence d'aspect (n! Configuration)

     2.2- Supprimer et Ajouter les arcs nécessaires

3- Fin si

4- Fin Pour
```

1- PJ = { *communication.print()*, *communication.print()* }

pj= *communication.print()*

2- vrai (pj se trouve (2) deux fois) et (2>1) et (les advices associées à pj de même type « after »)

2.1- le nombre des configurations possible = 2 ! = 2.

2.2- les arcs supprimer/ajouter. Supprimer l'arc (3-4) et ajouter les deux arcs suivants touchant le nœud 3 :

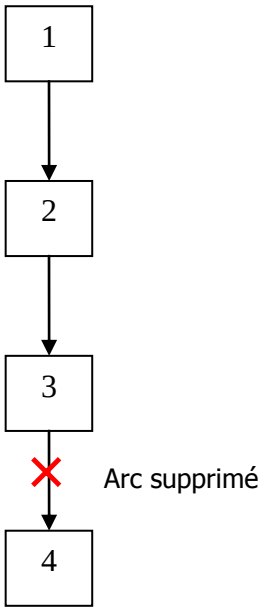
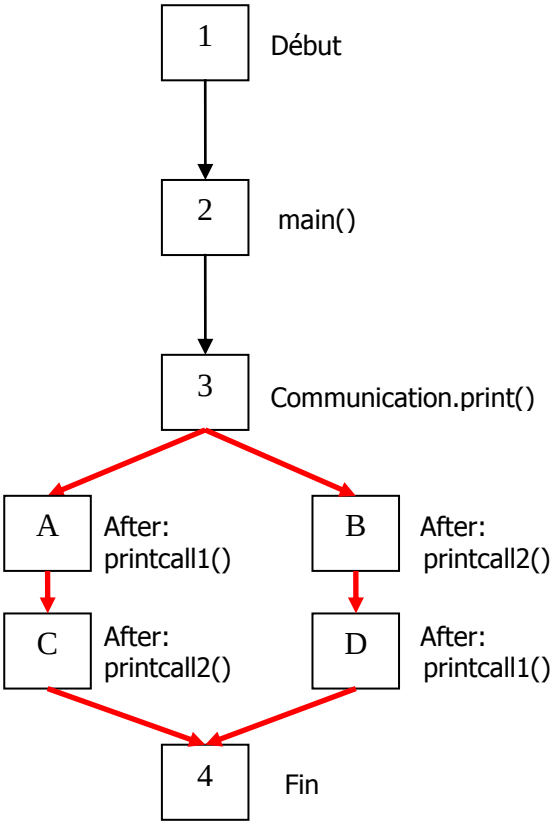
Arc1(3-4) : entre la méthode *communication.print()* et l'advice after *printcall1()*, cette opération touche le nœud 3.

Arc2(4-6) : entre l'advice after *printcall1()* et l'advices after *printcall2()*, première configuration dont l'ordre de tissage est : *questionaspect*, *learnaspect*.

Arc3(3-5) : entre la méthode *communication.print()* et l'advice after *printcall2()*, cette opération touche le nœud 3.

Arc4(5-7) entre l'advice after *printcall2()* et l'advices after *printcall1()*, deuxième configuration dont l'ordre de tissage est *LearnAspect*, *QuestionAspect*.

3- 4- fin

Graphe du code de base	**Le graphe complet après le tissage des nouveaux arcs**
	

c- Analyse. L'arc (3-4) est supprimer et remplacer par les deux sous chemins (3,A,C) et (3,B,D). Les chemins possibles sont :

Chemin 1 : 1,2,3,A,C,4

Chemin 2 : 1,2,3,B,D,4

Le chemin 2 détecte le défaut lié à la précédence incorrecte des aspects (le modèle de défaut 2).

5.2 La détection du modèle de défaut 5 (Mauvaise orientation du flux de contrôle)

Pour bien illustrer le fonctionnement de notre technique, nous donnons dans cette section une application exemple constituée de deux classes et d'un aspect. Il s'agit de transactions sur un compte bancaire.

L'application : bank

```
public class Account {
  int id; //account id
  Database db; //associated database
  float getBalance(){
    return db.query("SELECT balance FROM accounts WHERE id="+id);
  }
  void setBalance(float b){
    db.update("UPDATE accounts SET balance="+b+"WHERE id="+id);
  }
  void credit(float amount){
    setBalance(getBalance()+amount);
  }
}
Public class principale {    // Classe principale
  Public static void main (string[] args) {
     Account account = new Account();
     account.credit(100);
  }
}
public class Database {
  float query(String sqlQuery){
    // lecture de la base de données
  }
  void update(String sqlQuery){
  // Écriture dans la base de données
  }
}
public aspect AutoLogAspect_CallF {
  pointcut creditMethodCallF():
                    cflow(call (* Account.credit(..)));
  before() creditMethodCallF(){
    System.out.println("Hello!");
  }
}
```

Dans cette application on utilise une primitive dynamique *cflow*, la désignation du point de coupure qui dispose une primitive dynamique ne peut être résoulue entièrement au moment du tissage. Ce point nécessite une évaluation des conditions au moment d'exécution.

a- Construction du graphe de contrôle du code de base. Le code de base de cet exemple est composé de trois classes : la classe *Account*, la classe *Database*, et la classe *Principale*. Avant de construire le graphe du flux de contrôle de l'application *bank*

il faux connaître le flux de l'appel à la méthode *Account.credit()*. Ce dernier est illustré dans la figure 4.1.

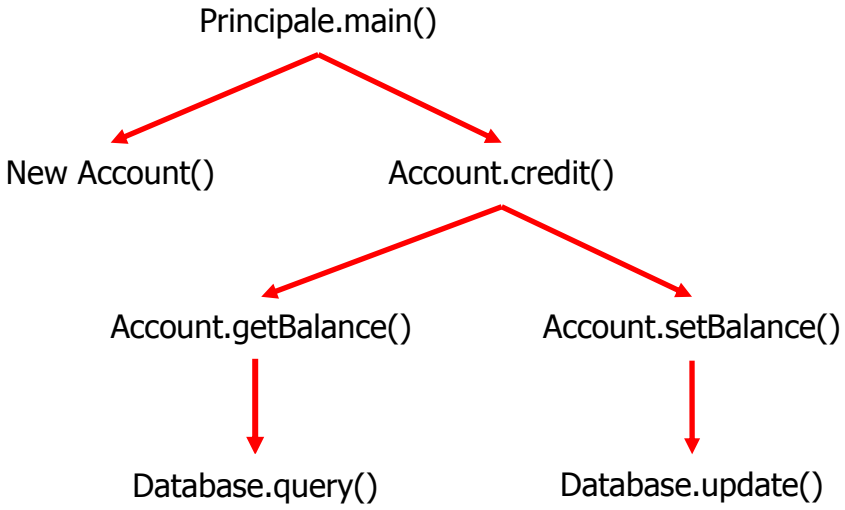
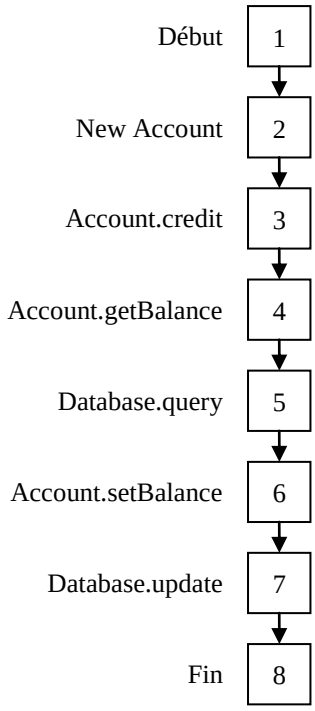


Figure 4.1 : Graphe de contrôle de l'appelle à la méthode Account.credit().

Le graphe de contrôle est donné par le tableau suivant.

Le programme	**Le graphe de contrôle**
1- Début 2- La méthode main 3- Account account = new Account(); 4- Account.credit	

Application de l'algorithme

```
1-Pour chaque aspect A faire
 2- Pour chaque point de coupure de l'aspect A faire
  3- Si point de coupure ≠ primitive dynamique alors
     4- Identifier les points de jointure dans le GFC associés au
     point de coupure
     5- Ajouter les arcs correspondants aux consignes tissées aux
     points de jointures
  6- Fin si
  7- Si point de coupure = primitive dynamique alors
                  /* pour obtenir tous les points de jointures
     8- Remplacer la condition de la primitive par vrai
     9- Identifier les points de jointure dans le GFC associés aux
     points de coupure
     10- Supprimer et ajouter les arcs correspondants aux advices
     tissées.
  11- Fin si
 12- Fin pour
13- Fin pour
```

1- pour l'aspect AutoLogAspect_CallF faire

2- pour le point de coupure creditMethodCallF() faire

3- faux

7- vrai

8- cflow (call (*Account.credit(..))) evaluée à vrai;

cflow (call (*Account.credit(..))) tout point de jointure dans le flux de contrôle d'une méthode credit de la classe Account, y compris l'appel à la méthode credit elle même.

9- à partir d'évaluation et du flot de contrôle donné par la figure 4.1, l'ensemble de tous les points de jointure possibles est { credit(), getBalance, setBalance, query, update}

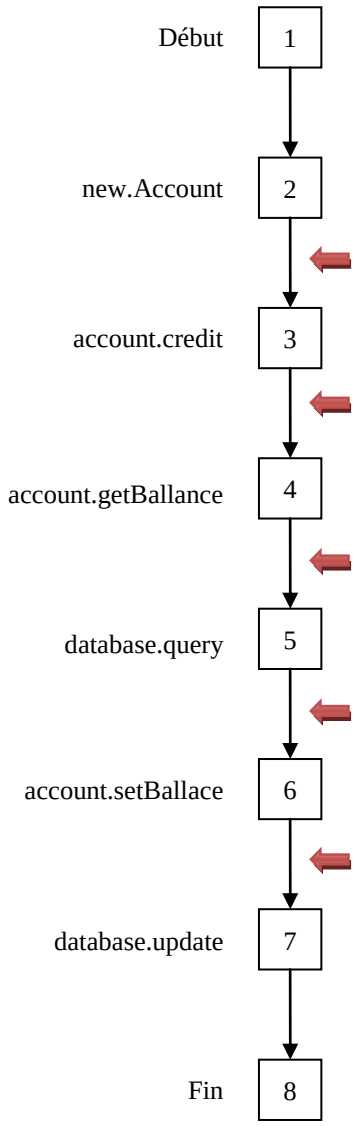
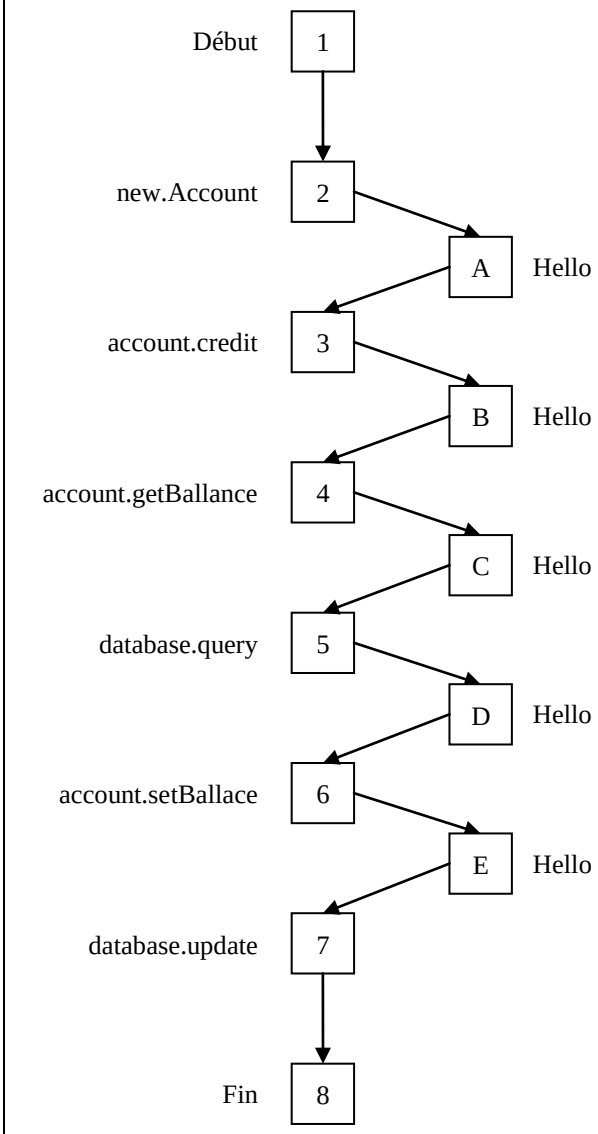
10- les arcs supprimer/ ajouter.

- l'Arc (2-3) supprimer et remplacer par (2-A) et (A-3).
- l'Arc (3-4) supprimer et remplacer par (3-B) et (B-4).
- l'Arc (4-5) supprimer et remplacer par (4-C) et (C-5).
- l'Arc (5-6) supprimer et remplacer par (5-D) et (D-6).
- l'Arc (6-7) supprimer et remplacer par (6-E) et (E-7).

11-fin si

12- fin pour

Ces modifications sont présentées dans le tableau suivant.

Le graphe du code de base	**Le graphe complet après tissage des nouveaux arcs**
	

b- Analyse. Dans cet exemple l'aspect possède un point de coupure qui utilise une primitive dynamique. La désignation de ce point de coupure ne peut être résolue entièrement au moment du tissage. En réalité ce type de points de coupure doit être évalué au moment de l'exécution.

Les points de jointures dans lesquelles il faut tisser l'advice *before() : creditMethodCallF()* sont : {*credit(), getBalance, setBalance, query, update*}. Vu la description de *cflow(call (* Account.credit(..))*, tout point de jointure dans le flux de contrôle de la méthode *credit()* de

la classe *Account*, y compris l'appel à la méthode *credit()* elle même. Autrement dit, c'est les points de jointure entrant et sortant mais au moment de l'exécution. Cette description conduit à des configurations différentes tout dépend de l'évaluation au moment d'exécution, on a alors les point de jointure suivants :

1- Tout point dans le flot de contrôle de la méthode crédit() de la classe Account

= les points de jointure entrants + sortants

= { getBalance, setBalance, query, update}

2- y compris l'appel à la méthodes crédit() elle même

= { crédit() }

5.3 Comparaison avec les approches existantes

Notre technique est structurelle et sa comparaison est significative avec les techniques structurelles. La comparaison a lieu principalement pour la détection des modèles de défauts proposés par R.Alexander et J.Biemman [1], et en particulier la précédence incorrecte des aspects (modèle de défaut 2), et la mauvaise orientation du flux de contrôle (modèle de défauts 5).

A partir de l'analyse des modèles de défauts et la comparaison des approches selon les modèles de défauts (voir 2.1.1 dans ce chapitre), nous concluons que les modèles de défauts 2 et 5 sont les modèles les plus difficiles à détecter. Le problème principal dans la détection de ces deux modèles de défauts est leur nature qui exige une évaluation au moment de l'exécution (évaluation dynamique).

En peut résumer ce problème en deux sous problèmes principaux, le premier concerne la nature statique de ces approches due à l'utilisation du compilateur d'AspectJ qui est un compilateur statique.

Le deuxième est que ces approches sont basées principalement sur les techniques de test orientées-objets et ce dernier paradigme n'a pas besoin d'une évaluation dynamique à cause de la nature statique des programmes orientés-objets.

Concernant la composition des aspects au sein du programme de base, nous pouvons dire que la POA est une approche de bas niveau, parce que la POA travaille directement au niveau du code source. Aucune vision de haut niveau d'abstraction n'est utilisée pour spécifier les aspects ainsi que le programme de base. Nous considérons une telle composition comme une intégration.

La structure interne du programme de base est complètement visible aux aspects. Par conséquent, il est difficile de faire évoluer le programme de base car un changement mineur de ce programme peut perturber la localisation des endroits où il faut tisser (attacher) les aspects. Ceci peut être vu clairement dans AspectJ où les méthodes du programme de base sont utilisées pour définir les endroits où attacher les aspects.

Concernant la composition d'aspects attachés au même point dans le programme de référence, nous pouvons distinguer deux manières :

i. Par l'ordonnancement des aspects, on ne peut pas avoir un contrôle très fin sur la composition des aspects. Ceci doit se baser sur l'hypothèse qu'on n'a pas besoin de coordonner les aspects et que les aspects sont orthogonaux.

ii. Par le tissage, la composition d'aspects peut être réalisée en entremêlant des instructions de ces aspects. Ceci permet un contrôle très fin sur la coordination des aspects.

Il est donc possible de travailler avec des aspects non-orthogonaux. AspectJ s'intéresse trop peu aux problèmes de composition des aspects. C'est pour cela qu'il prend l'ordonnancement comme étant le moyen de composer les aspects attachés à un même point de jonction. Pour ordonner les aspects, AspectJ propose quelques règles d'ordonnancement assez simples. Ceci n'est pas suffisant car il est fréquent que les aspects ne soient pas orthogonaux. Par exemple, un aspect de persistance peut être lié à un aspect de transaction.

Notre technique comble des insuffisances par la construction de l'algorithme GCC (graphe de contrôle complet) qui se base sur 3 points principaux :

1. Le tissage des tests d'aspects au graphe du flux de contrôle du code de base.
2. Évaluation complète des primitives dynamiques.
3. Définition de l'ordre de tissage d'aspects par la définition de toutes les configurations de tissage possible.

Conclusion

Dans ce chapitre nous avons présenté l'essentiel de notre approche de test et nous avons montré ses capacités par rapport aux autres approches. En particulier, sa capacité à détecter les modèles de défauts 2 et 5 qui ne sont pas détectés par les autres approches.

La technique que nous avons proposée est structurelle, et elle se base sur l'évaluation des points de jointure dynamiques, pour la détection des modèles de défauts 2, et 5.

Dans le chapitre suivant, nous exposerons les aspects relatifs à la conception et à la mise en œuvre de l'environnement de test que nous avons réalisé.

CHAPITRE 5
CONCEPTION ET MISE EN OEUVRE

Ce chapitre apporte des éléments de réponse à la concrétisation de la technique proposée dans ce mémoire. Dans un premier temps, nous présentons, moyennant des artéfacts UML, les éléments essentiels de l'environnement support de notre technique. Dans un second temps, nous donnons un aperçu sur l'environnement et son utilisation telle que vue par un utilisateur externe.

1 Présentation conceptuelle de notre projet

La phase clé de notre projet est la construction du graphe de contrôle complet pour le test d'un programme orienté-aspects, via les trois axes cités dans le chapitre précédent. Cette phase comprend notamment, la construction du graphe de contrôle du code de base, l'application de l'algorithme GCC pour compléter le graphe construit, puis l'extraction des chemins.

Une analyse statique du programme sous test est opérée. Les étapes détaillées sont les suivantes :

Etape 1. Dans cette étape, il s'agit d'extraire les différentes unités de test (les classes et les aspects) à partir du programme initial puis de séparer les méthodes de chaque classe et les advices de chaque aspect, pour aboutir à une structure hiérarchique des composants du programme sous test.

Etape 2. Elle consiste à générer un graphe de contrôle à partir du programme de base (l'ensemble des classes). Cette transformation s'effectue à l'aide d'une analyse statique du code source, en déterminant les blocs d'instructions qui présentent des nœuds de ce graphe, et des arcs qui les relient.

Etape 3. Le but de cette étape est de compléter le graphe déjà construit (Etape 2) par les éléments transversaux du programme. Cette étape consiste à analyser le code des aspects.

Etape 4. Cette étape consiste à construire l'ensemble des chemins à partir du graphe produit dans l'étape précédente.

Etape 5. Dans cette étape, une recherche des chemins associés aux points de jointures dynamiques est opérée afin de présenter la détection des modèles de défauts (2 et 5).

2 La modélisation UML du projet

2.1 Les cas d'utilisation

Deux concepts fondamentaux interviennent dans la modélisation par les uses cases :

1- Les acteurs ou utilisateurs du système, ils ont une bonne connaissance des fonctionnalités du système. Ils sont extérieurs au système et peuvent être des humains, ou des logiciels.

2- Les uses cases proprement dit qui décrivent les utilisations du système et ses objectifs.

Il existe plusieurs façons de décrire les cas d'utilisation, une de ces façons est la description textuelle. Pour notre projet la description textuelle de chaque cas d'utilisation est donnée ci-après et la description graphique est donnée par la figure 5.1.

a- Génération du graphe du code de base

1. L'utilisateur introduit le chemin du programme a tester.
2. Le système récupère ce chemin.
3. Le système construit et affiche le graphe.

b- Génération du graphe complet (constituant le tissage des aspect)

1. L'utilisateur demander la génération du graphe complet.
2. Le système applique l'algorithme gcc pour la génération du graphe demandé.

c- Affichage des chemins associés aux points de jointure dynamiques

1. L'utilisateur demande l'affichage des chemins associés aux points de jointure dynamiques.
2. Le système affiche ces chemins

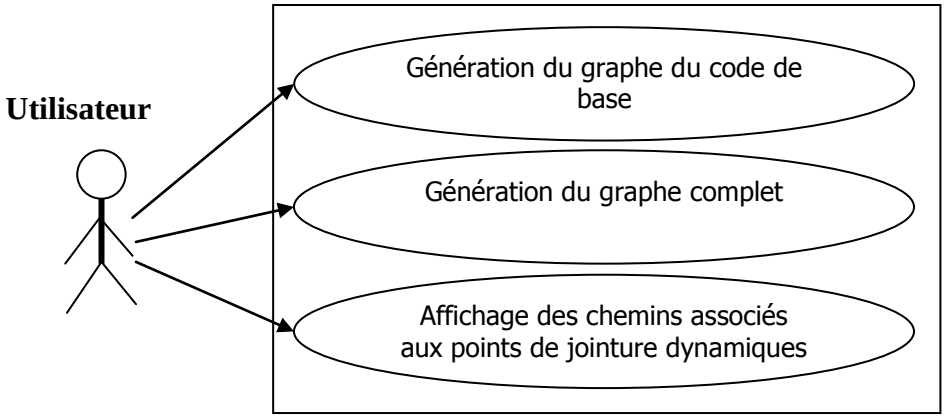


Figure 5.1 : Les cas d'utilisation dans notre projet

2.2 Diagramme de séquence

Le diagramme de séquences est la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique. Il montre les interactions dans le cadre du scénario d'un cas d'utilisation. Dans la figure 5.2, nous donnons un diagramme de séquence qui représente les interactions globales de notre système et un utilisateur testeur.

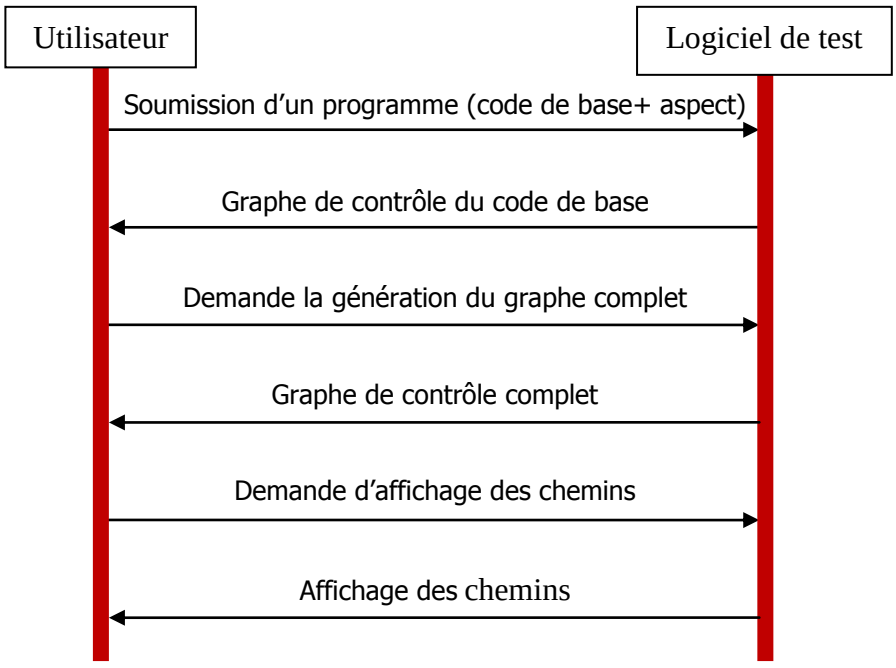


Figure 5.2 : Diagramme de séquence

Au début l'utilisateur introduit le programme qu'il souhaite tester. Le logiciel de test récupère ce programme, cherche les classes et construit le graphe du code de base.

L'utilisateur demande alors la génération du graphe de contrôle complet (avec le tissage des aspects) et le logiciel de test applique l'algorithme et construit le graphe complet.

A la fin l'utilisateur demande l'affichage des chemins associés aux points de jointure dynamiques et le logiciel de test affiche les chemins demandés.

2.3 Diagramme d'activité

Le diagramme d'activité est un diagramme comportemental d'UML, permettant de représenter le déclenchement d'événements en fonction des états du système et de modéliser des comportements parallélisables (multi-threads ou multi-processus).

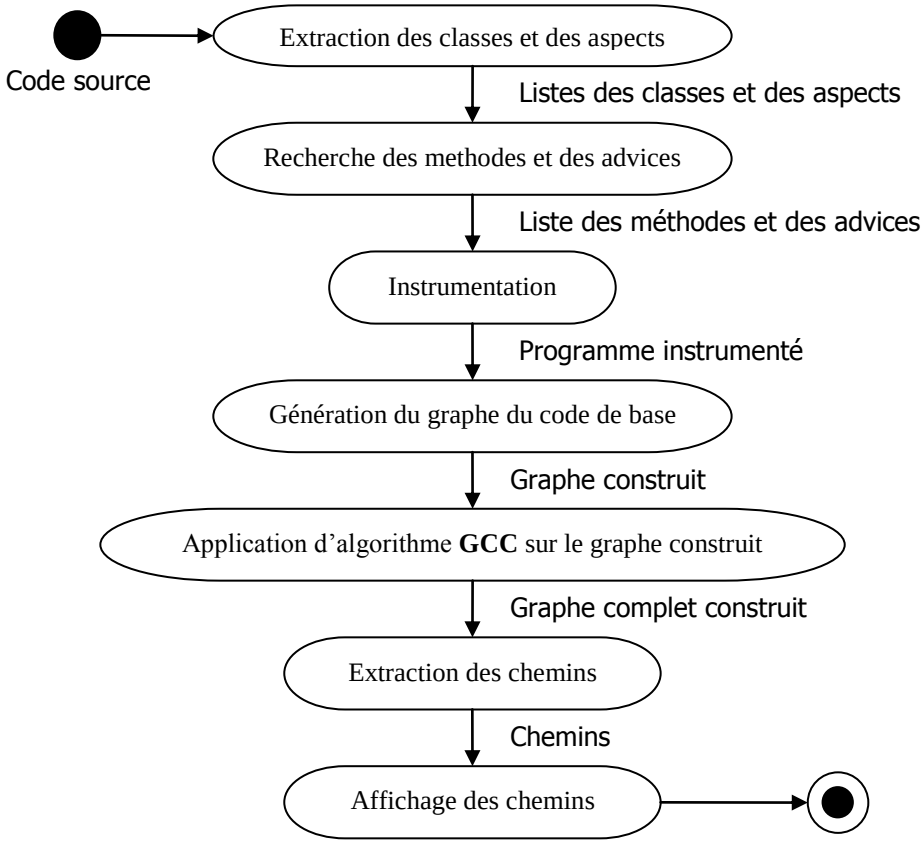


Figure 5.3 : Diagramme d'activités

1. Initialement le système effectue une extraction des différentes classes et aspects qui composent le programme sous test.

2. Il passe par la suite à la recherche de leurs unités (méthodes des classes et advices des aspects).

3. L'étape suivante du système est une instrumentation, qui consiste à ajouter des instructions au programme de base qui présente la partie fonctionnelle (l'ensemble des classes) sous test, afin de marquer les blocs d'instructions.

4. Après la sélection d'une section instrumentée (la méthode main()), le système réalise une génération du graphe, dont il formule la structure à partir de la succession des nœuds marqués, reliés par des arcs.

5. Le système applique l'algorithme GCC qui comprend la sélection de chaque bloc instrumenté (point de coupure). Le système précise les nœuds sensibles aux aspects parmi l'ensemble des nœuds constituants le graphe de contrôle du code de base et selon le type de l'advice (avant ou après). Le système supprime les arcs et ajoute de nouveaux arcs selon l'algorithme de la construction du graphe du flux de contrôle complet afin d'obtenir le graphe complet du programme sous test.

6. Le système fait une extraction des chemins et affiche les chemins associées aux points de jointure dynamiques.

7. Le système affiche les chemins trouvés.

2.4 Diagramme de classe

Le diagramme de classes est un schéma pour présenter les classes et les interfaces d'un système ainsi que les différentes relations entre celles-ci. Nous l'utilisons pour montrer les entités essentielles de notre système (Fig. 5.4).

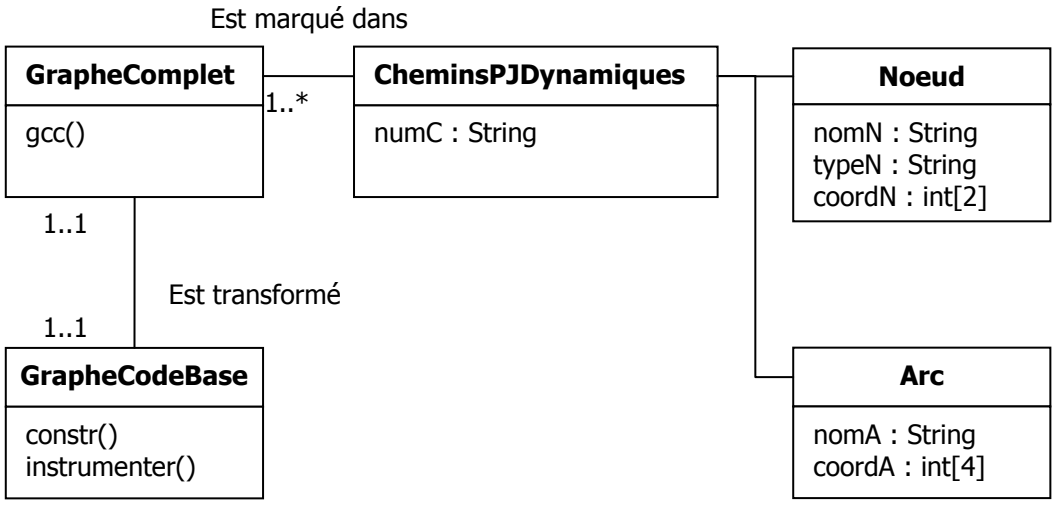


Figure 5.4 : Le diagramme des classes

La structure de notre système apparaît sous forme d'un ensemble de classes et de relations qui les relient.

La classe *GrapheCodeBase* permet la construction du graphe à partir du programme sous test par la méthode *Construir()*. Cette étape est suivie d'une phase instrumentation du code, qui est modélisée par la méthode *instrumenter()*. Le graphe du code de base est transformé en graphe complet.

La classe GrapheComplet modélise le graphe complet. Par l'application de l'algorithme gcc(), le graphe sera composé d'un ensemble de chemins.

La classe CheminsPJDynamiques modélise les chemins qui se caractérisent par un numéro (l'attribut numC) et sont composés d'un ensemble de nœuds qui sont des PJ dynamiques et d'arcs les joignant.

La classe Nœud modélise les nœuds qui correspondent aux blocs d'instructions dans le programme sous le test, et des PJ. Un nœud se caractérise par un nom (l'attribut nomN), un type (l'attribut typeN) et des coordonnés graphiques (l'attribut coordN).

La classe Arc modélise le lien entre deux nœuds adjacents dans un chemin. L'arc se caractérise par un nom (l'attribut nomA), et les coordonnées graphiques (l'attribut coordA).

3 Quelques considérations pratiques

Les programmes à tester sont des programmes orientés-aspects. Pour illustrer le fonctionnement de notre technique, nous avons choisit le langage *JCreator* pour le programme principal et le langage AspectJ version 1.5 pour les aspects.

JCreator est un outil de développement rapide, efficace (car entièrement écrit en C++ [34]). C'est un outil qui convient aussi bien aux programmeurs qu'aux experts en Java. JCreator fournit à l'utilisateur un large éventail de fonctionnalités telles que la gestion des projets, le débogage, édition avec coloration syntaxique, assistance et interface utilisateur personnalisables. Avec Jcreator on peut directement compiler ou lancer un programme Java sans activer le document principal en premier. La figure 5.5 donne un aperçu sur l'environnement Jcreator. Notons, cependant, qu'il n'intègre aucun outil qui permet de créer des applications graphiquement.

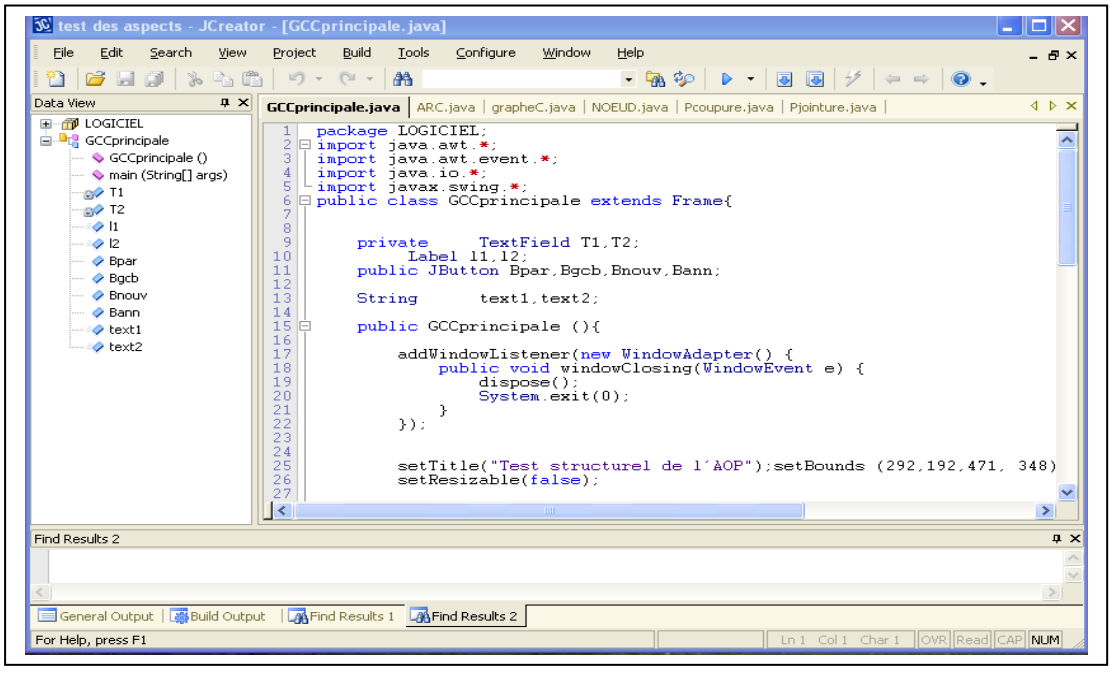


Figure 5.5 : Fenêtre de l'environnement JCreator

De son côté AspectJ version 1.5 est l'outil le plus utilisé actuellement dans le développement par aspects et sa version 1.5 est la plus récente et est compatible avec les nouvelles versions de java. Le code d'AspectJ peut être ajouté à un programme existant (remplacer un tas de code ou lui ajouter des contrôles ou traitement de débogage).

4 Mise en œuvre de notre approche

4.1 Fonctionnement de notre système

La mise en œuvre a pour objectif de définir un procédé approprié pour l'application de l'algorithme GCC. L'environnement support de l'algorithme GCC analyse statiquement le code pour construire le graphe complet correspondant. Il inclus deux algorithmes complémentaires, le premier pour la construction du graphe du code de base, et le second pour la construction du graphe complet par application de l'algorithme GCC.

Construction du graphe du code de base. Cette fonctionnalité consiste à analyser statiquement le code de base dans le but de formuler le squelette du graphe, en spécifiant les blocs d'instruction et en déterminant leur succession. L'algorithme est donné dans ce qui suit.

```
Entrées : Le programme sous test.
Sorties : Le graphe du code de base.
Début
  Parcourir et instrumenter le programme sous test afin de le
fractionner en méthodes
  Pour chaque entité (une méthode) faire
    Parcourir l'entité, trouver et marquer les blocs d'instruction
(les nœuds du graphe) en instrumentant cette entité. Il y a trois
types possibles de nœuds   les boucles, les appels de méthodes et
les portions linéaires du code (bloc simple) ;
    Trouver les arcs joignant ces nœuds, de sorte que, chaque nœud
soit relié avec son suivant et un traitement particulier ait lieu
pour un type spécifique de nœud ;
    Calculer les coordonnés graphique des nœuds et des arcs ;
  Enregistrer les éléments du graphe trouvés précédemment (nœud,
arcs, coordonnés) dans une structure de donnée ;
  Fin pour
Fin
```

Construction du graphe complet. Cette fonctionnalité est donnée par ce qui suit.

```
Entrée : Le graphe du code de base (le résultat de l'algorithme
précédent)
Sorties : Le graphe complet.
   Les chemins sensibles aux points de jointure.
Début
  Pour chaque aspect faire
    Pour chaque point de coupure faire
      si le point de coupure ne contient pas de primitives
      dynamiques alors
        Identifier les points de jointure dans le GFC
        associés au point de coupure.
        Ajouter les ARCs correspondants aux advices tissées
        aux points de jointure.
      Fin si
      si le point de coupure contient une primitive dynamique
      alors
        Remplacer La condition de la primitive par vrai
        Identifier les points de jointure dans le GFC
        associés aux points de coupure
        Supprimer et Ajouter les arcs correspondants aux
        advices tissées.
      Fin si
    Fin pour
  Fin pour
  Pour chaque point de jointure appartenant au tissage de plus
  d'un aspect faire
    Crée toutes les configurations possibles de la précédence
    d'aspect ( n! Configurations)
    Ajouter les arcs nécessaires
  Fin pour
Fin.
```

4.2 L'instrumentation

L'instrumentation consiste en l'ajout des instructions dans le code du programme sous test (dans ce cas le code de base en particulier). Dans notre projet, nous avons utilisé l'instrumentation quand on a généré le graphe du code de base. Nous avons ajouté dans le code des instructions qui nous permettent de marquer les nœud du graphe.

4.3 Le programme sous test

Le programme sous test est les deux applications : « communication et bank » utilisées comme exemple de l'application de la technique proposée dans le chapitre 4, écrite en JCreator.

5 Navigation dans l'application

Dans l'apparence externe de notre système nous avons principalement trois interfaces « l'interface graphe du code de base », « l'interface du graphe complet », « l'interface des chemins », et une fenêtre d'accueil et de saisie du chemin d'accès au programme sous test.

1- l'application « communication » La détection du modèle de défaut 2

La fenêtre d'accueil. Cette fenêtre est illustrée à la figure 5.6. Le bouton *Grapphecode de base* permet la construction du graphe du code de base et l'affichage de celui-ci dans la fenêtre Graphe du code de base

Figure 5.6 : La fenêtre d'accueil

La fenêtre du graphe du code de base. La fenêtre du graphe du code de base affiche le graphe du code de base et donne la main à l'utilisateur pour afficher la fenêtre du graphe complet via le bouton *GRAPHE COMPLET* (Fig 5.7).

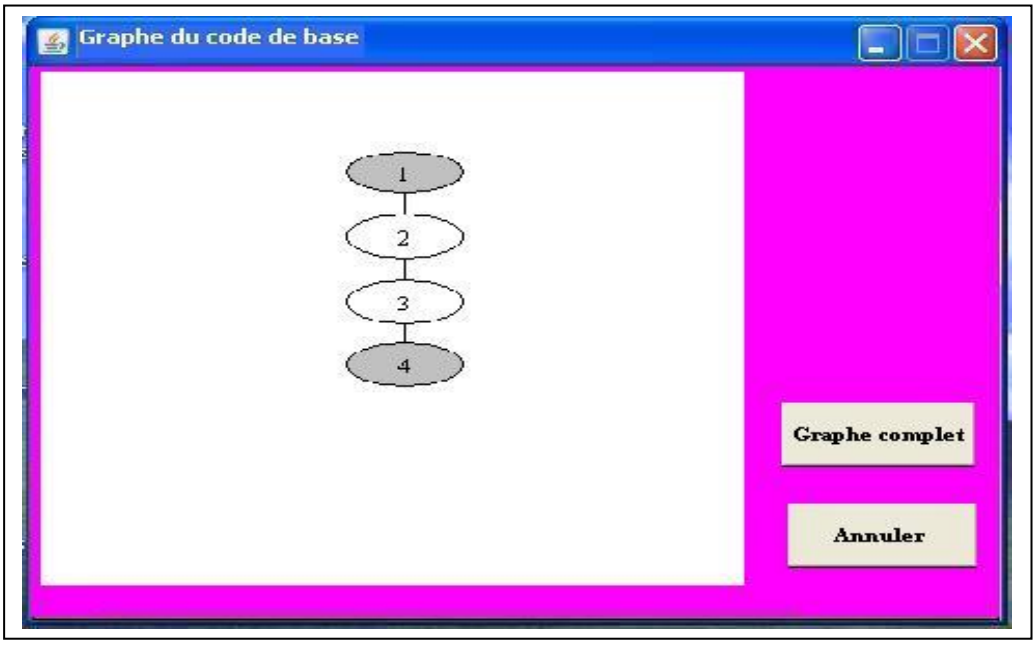


Figure 5.7 : Fenêtre affichant le graphe du code de base (application communication)

La fenêtre graphe complet. La fenêtre de la figure 5.8 affiche le graphe complet après tissage des aspects et donne la main à l'utilisateur pour afficher la fenêtre des chemins liées aux points de jointures via le bouton *chemins*.

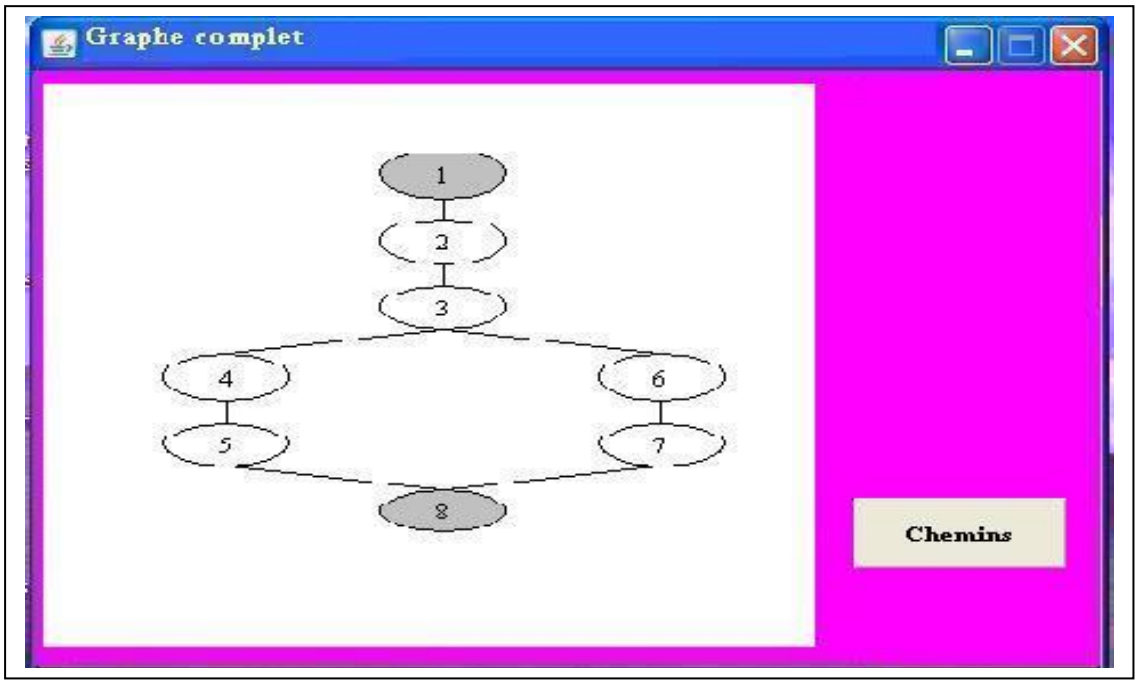
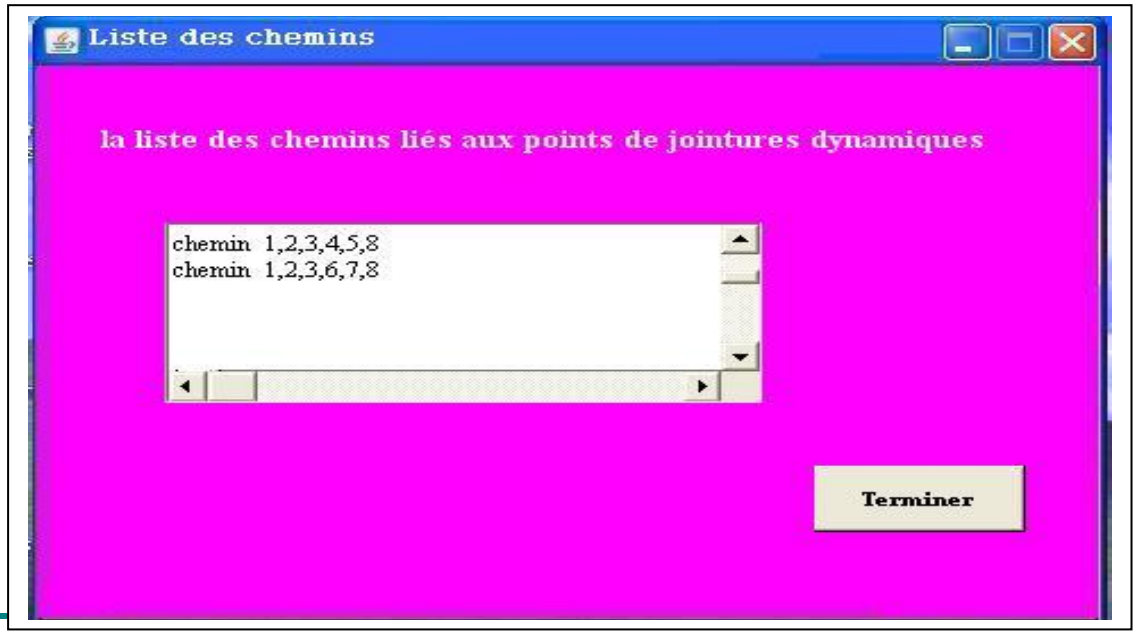


Figure 5.8: Graphe complet après tissage des aspects (application communication)

La fenêtre liste des chemins. La fenêtre liste des chemins affiche la liste des chemins liés aux points de jointures dynamiques (Fig 5.9).



Fenêtre 5.9 : LIste des chemins liés aux points de jointures dynamiques (application communication)

2-l'application « bank » la détection du modèle de défaut 5

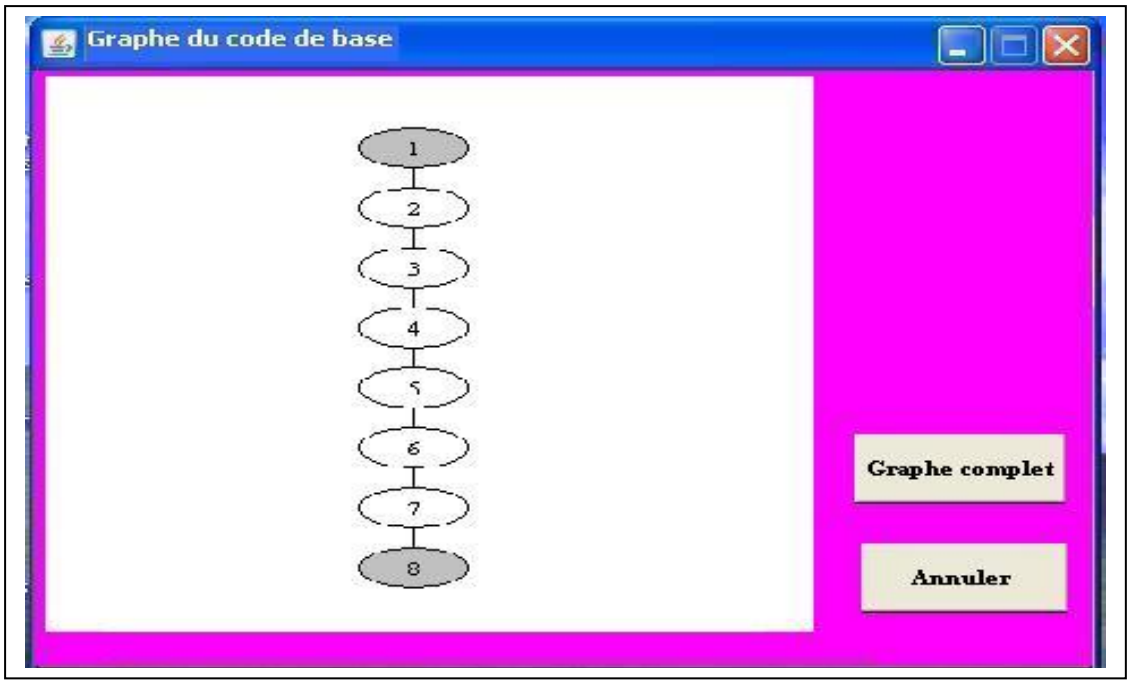


Figure 5.10 : Fenêtre affichant le graphe du code de base (application bank)

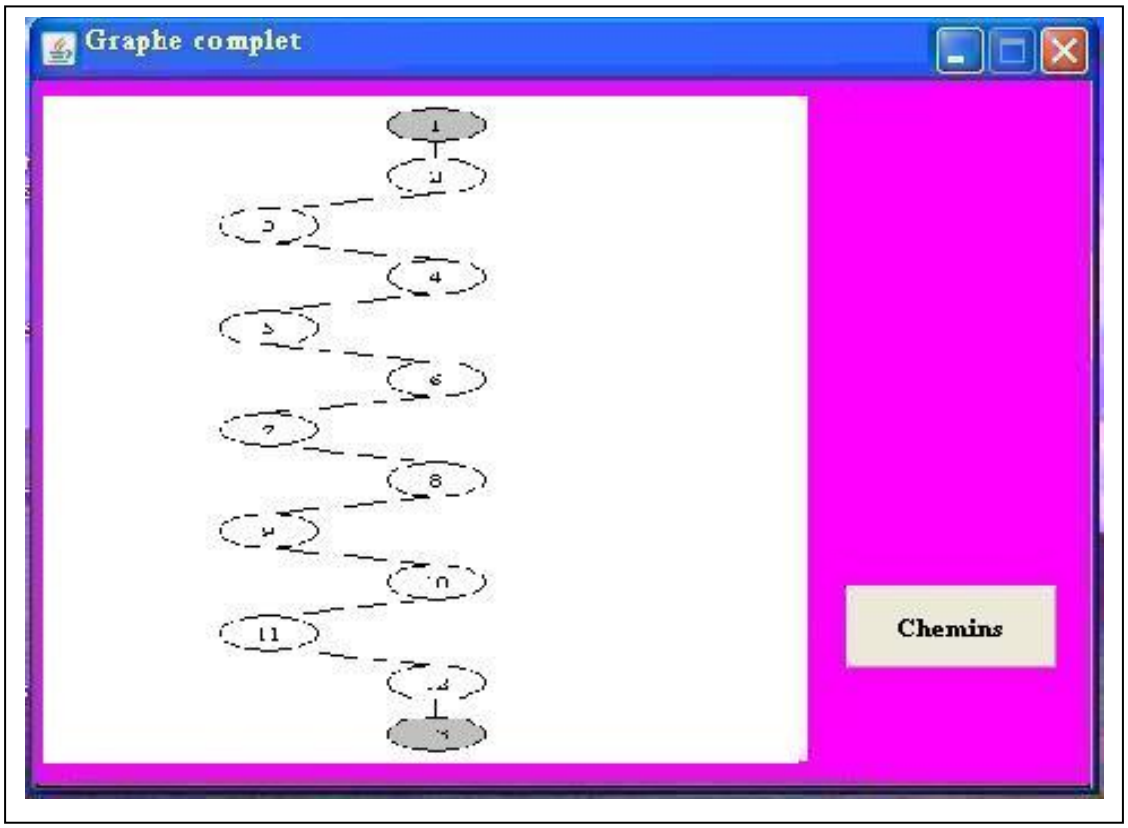
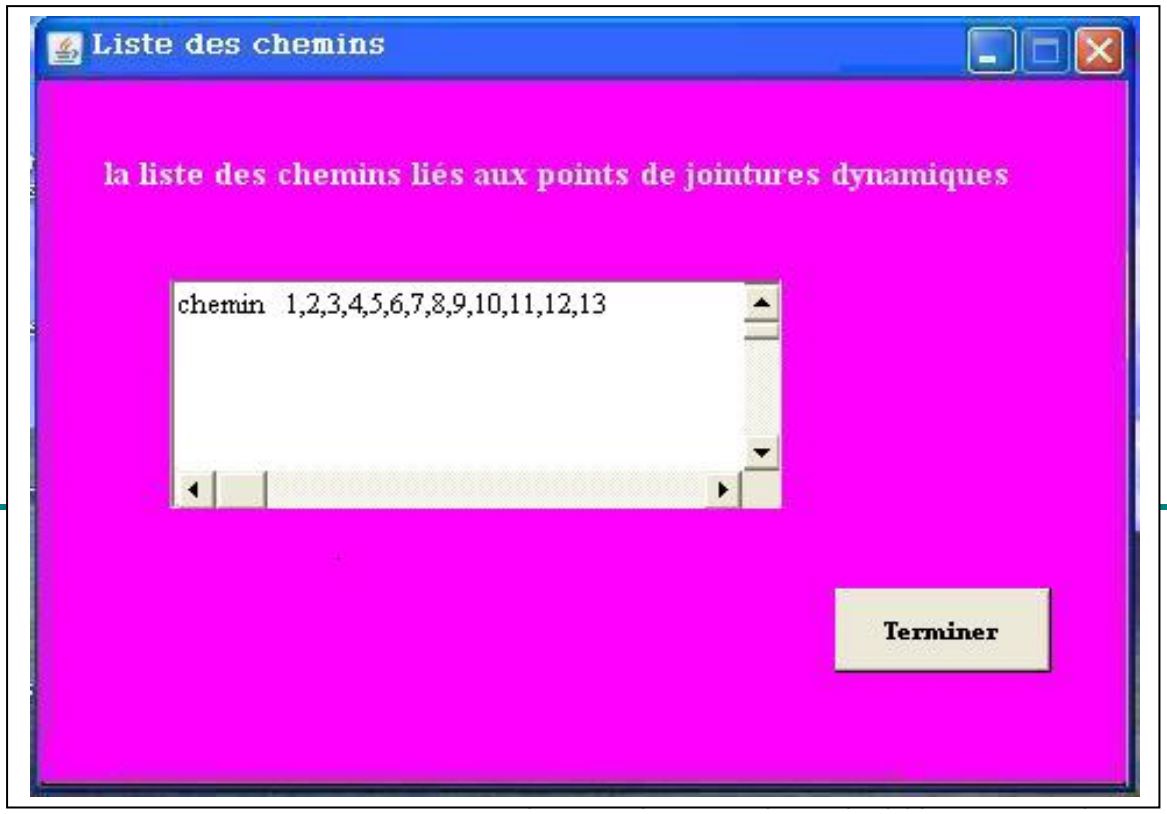


Figure 5.11: Graphe complet après tissage des aspects (application Bank)



Conclusion

Dans ce chapitre nous avons présenté le système qui supporte notre approche de test. Nous avons fait usage des diagrammes de la notation UML pour mettre en évidence les aspects conceptuels les plus importants de notre travail.

Par la suite, nous avons parlé de l'environnement de développement et le langage choisi dans la programmation. Puis, nous avons présenté le fonctionnement de la technique proposée et en particulier l'algorithme GCC. En dernier lieu nous avons donné un aperçu sur les interfaces réalisées.

CONCLUSION ET PERSPECTIVES

Le test apparaît aujourd'hui comme le moyen principal pour la validation du fonctionnement d'un logiciel. Il a pour objectif d'examiner ou d'exécuter un programme dans l'intention d'y détecter un maximum de défauts et de mettre en évidence des points éventuels ou le comportement n'est pas celui attendu. Le test demeure un moyen efficace pour estimer la confiance vis-à-vis d'un logiciel ou d'un composant logiciel.

L'activité du test, omniprésente tout au long du cycle de vie du logiciel, est mise en œuvre par différentes techniques permettant la validation des différentes étapes du développement.

Une bonne technique de test doit avoir une capacité suffisante à détecter les défauts qui peuvent affecter un logiciel, en particulier, lorsqu'on connait préalablement les modèles de défauts qui affectent les implémentations d'un paradigme donnée. C'est le cas pour le nouveau paradigme de programmation, qui est la programmation orientée–aspects. Cette approche apporte une solution nouvelle et élégante aux problèmes d'éparpillement et d'enchevêtrement qui existent dans la plupart des implémentations orientées-objets des systèmes informatiques.

Cependant, malgré que ce paradigme fournisse un bon mécanisme de modularité il pose cependant des problèmes parmi lesquels, la difficulté de vérification. Des approches on été proposées actuellement pour le test des programmes orientés-aspects. Cependant, elles souffrent d'insuffisances qui les rendent incapable d'être des références solides pour le test des programmes orientés-aspects.

Cette conclusion vient suite à notre étude comparative entre ces approches. Nous avons mené cette comparaison en se basant sur deux facettes. Conformément à la première, nous avons fait ressortir les aptitudes et les insuffisances de chacune, et conformément à la deuxième, nous avons utilisé six modèles de défauts spécifiques aux approches orientées-aspects.

L'analyse des approches en fonction de ces modèles de défauts nous a permis de conclure que les défauts qui affectent les programmes orientés-aspects se divisent en deux classes. Des défauts statiques détectables à la compilation (les défauts 1,3,4,6 sont détectables avant ou pendant le tissage) et des défauts dynamiques nécessitant une évaluation au moment de

l'exécution (les défauts 2,5). Les programmes orientés-aspects étant similaires sur plusieurs points aux programmes orientés-objets, les défauts statiques sont détectables à partir des approches de test orientées-objets avec certain spécialisation pour l'entité *aspect*. Ainsi les approches de test orientées-aspects qui sont des extensions de celles orientées-objets ont la capacité de détecter certains défauts statiques.

Le problème se pose alors pour les défauts dynamiques dont la détection n'est pas possible par les approches de test orientées-aspects existantes.

La technique de test structurelle que nous avons proposé pour le test des programmes orientés–aspects est basée sur les points de jointures dynamiques. Elle a comme but de détecter les modèles de défauts dynamiques (2 et 5) via un algorithme que nous avons nommé GCC (Graphe de Contrôle Complet).

Pour la phase d'analyse et de conception de l'environnement support de l'algorithme GCC, nous avons opté pour la notation UML qui est devenue une référence incontournable dans le domaine du génie logiciel. Pour la concrétisation de ce travail, nous avons utilisé le langage de programmation JCreator.

Concernant les perspectives de notre travail, elles sont d'ordre pratique. Il reste un travail mineur pour la complétude de l'approche dans la détection des défauts statiques qui sont déjà détectés par d'autres approches. La complétude peut être effectuée dans le cadre d'une hybridation de la technique avec des techniques existantes. Sur un autre plan, nous comptons tester notre approche sur des exemples plus consistants et réels.

RÉFÉRENCES

[1] R. Alexander, J. Bieman and A. Andrews, “Towards the Systematic Testing of Aspect-Oriented Programs”, Technical Report CS-4-105, Department of Computer Science, ColoradoState University, Fort Collins, Colorado, 2004.

[2] P. Anbalagan & T.Xie, “Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs”, Workshop on Mutation Analysis (MUTATION 06), Raleigh, November 2006.

[3] J. Baltus, "La Programmation Orientée Aspect et AspectJ : Présentation et Application dans un Système Distribué", Mini-Workshop: Systèmes Coopératifs. Matière Approfondie, Institut d’informatique, Namur, 2001.

[4] M. L. Bernardi & G. A. Di Lucca, "Testing Aspect Oriented Programs: an Approach Based on theCoverage of the Interactions among Advices and Methods," in Proceedings of the 6th InternationalConference on the Quality of Information and Communications Technology.QUATIC07. , 2007, pp. 65-76

[5] G. Denaro and M. Monga, “An Experience on Verification of Aspect Properties”, Proceedings of the International Workshop on Principles of Software Evolution (IWPSE01), Vienna, Austria, 2001.

[6] F. Duclos, "Environnement de Gestion de Services Non Fonctionnels Dans les Applications a Composants", Doctoral dissertation, Université Joseph Fourier, Octobre 2002.

[7] M. C. Gaudel et al., "Précis de Génie Logiciel", Masson, 1996.

[8] I. Sommerville, "Software engineering", Addison Wesley, 2007.

[9] T. Gil, "Conception orientéeaspect, version 2.1", DotnetGuru, 2007, http://www.lulu.com/content/830955

[10] IEEE Std 610.12-1990, “IEEE Standard Glossary of Software Engineering Terminology –Description”, IEEE, 1991.

[11] B. Legeard « cours test logiciel » laboratoire d’informatique de l’université de France

[12] R. Pawlak, et al., "Programmation orientée aspect pour Java / J2EE", édition Eyrolles, 2004.

[13] C. Oriat, "Tutoriel JML/Jartege", LSR-IMAG, Grenoble, Mai 2003, http://www-lsr.imag.fr/Les.Personnes/Catherine.Oriat/TutorielJML/

[14] N. Ubayashi & T. Tamai, “Aspect-Oriented programming with model checking”, AOSD’02, First international conference on aspect oriented software development, New york, USA,.2002.

[15] E. Wawszczyk, “Introduction à AOP (Aspect-Oriented Programming) avec le framework Spring”, décembre 2007, http://ewawszczyk.developpez.com/tutoriel/java/spring/aop/

[16] D. Xu, W. Xu, V. Goel and K. Nygard, "Aspect Flow Graph For Testing Aspect-Oriented Programs", Proceeding of the 8th IASTED International Conference on software Engineering and Applications. Oranjestad, Aruba (Caribbean), august 29-31, 2005

[17] T. Xie, J. Zhao, D. Marinov, and D. Notkin. “Detecting redundant unit tests for AspectJ programs”, In ISSRE'06: Proceedings of the 17th International Symposium on Software Reliability and Engineering, pages 179-190, 2006

[18] 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.

[19] G. Xu “A Regression Tests Selection Technique For Aspect-Oriented Programs”, Third Workshop on Testing Aspect-Oriented Programs (WTAOP'07), March 2007, Vancouver, British Columbia, Canada.

[20] 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.

[21] J. Zhao. “Data-flow-based unit testing of aspect-oriented programs”. In Proc. of the 27th COMPSAC, page 188. IEEE Computer Society, 2003.

[22] C. Zhao & R. Alexander, “Testing aspect-oriented programs as object-oriented programs”, Third Workshop on Testing Aspect-Oriented Programs (WTAOP'07), March 2007, Vancouver, British Columbia, Canada.

[23] C. Zhao and R. T. Alexander, “Testing AspectJ Programs using Fault-Based Testing”, Third Workshop on Testing Aspect-Oriented Programs (WTAOP'07), March 2007, Vancouver, British Columbia, Canada.

[24] J. Zhao, T. Xie, and N. Li, “Towards Regression Test Selection for Aspect-Oriented Programs”. In Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP‘06), Portland, Maine, pp. 21-26, July 2006.

[25] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, “The Art of Software Testing”, Second Edition, Wiley, 2004

[26] L’AFCIQ Agence Française de Contrôle Industriel de la Qualité

[27] www.apt.gc.ca/ap11140F.asp?pId=262

[28] http //atlastagcollector.in2p3.fr 8080/extra/bdsec/doc/PAQLv1.2.doc

[29] http //fr.wikipedia.org/wiki/AspectJ

[30] http //fr.wikipedia.org/wiki/Logiciel

[31] http //fr.wikipedia.org/wiki/Sp%C3%A9cification_(informatique)

[32] http //fr.wikipedia.org/wiki/Test_(informatique)#Qualit.C3.A9_et_Test

[33] http //jac.objectweb.org/docs/introduction-fr.html

[34] http //java developpez.com/faq/jcreator/ ?page=DECOUVRIR # DECOUVRIR quoi

[35] http //vieln.e-supinfo.net/my/aop/chapitre1.aspx?print=1

[36] http //xpose.avenir.asso.fr/viewxpose.php?site=36&subpage=/contents/presentation.html

[37] www.afis.fr/praout/eval/eval2.htm

[38] www-calfor.lip6.fr/~vmm/Enseignement/DESS/Test/Cours/C34.pdf

[39] www.clever-age.com/veille/blog/aop-dans-les-applications-java.html

[40] www.fr-aspectj.html

[41] www.futura-sciences.com/fr/comprendre/glossaire/definition/t/high-tech-1/d/logiciel_561/

[42] www-ic2.univ-lemans.fr/~alissali/Enseignement/Polys/ GL/node5.html

[43] www.infeig.unige.ch/support/se/lect/prg/tst/node8.html

[44] www.infeig.unige.ch/support/se/lect/prg/tst/node14.html

[45] www.laboiteaprog.com/article-44-4-genie_logiciel_cycle_ du_developpement logiciel

[46] www-list.cea.fr/labos/fr/LSL/test/docs/fr/memoirePM.pdf]

[47] www-list.cea.fr/labos/fr/LSL/test/index.html

[48] www-lsr.imag.fr/users/Catherine.Oriat/TutorielJML/tutoriel_1.html

[49] www.model-based-testing.org

[50] http //www.scriptol.fr/programmation/aspectj.php

[51] http //www.scriptol.org/fr-aspectj.html