Analyses Automatiques pour le Test de Programmes Orientés Aspect

Romain Delamare

▶ **To cite this version:**

Romain Delamare. Analyses Automatiques pour le Test de Programmes Orientés Aspect. Génie logiciel [cs.SE]. Université Rennes 1, 2009. Français. NNT : . tel-00512178

HAL Id: tel-00512178
https://theses.hal.science/tel-00512178

Submitted on 27 Aug 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N˚d'ordre : 4030 **ANNÉE 2009**

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Romain Delamare

préparée à l'IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
Composante Universitaire : IFSIC

Analyses automatiques pour le test de programmes orientés aspect

Thèse soutenue à Rennes le 2 décembre 2009

devant le jury composé de :

Thomas JENSEN
Directeur de Recherche au CNRS / *Président*
Laurence DUCHIEN
Professeur à l'Université de Lille 1 / *Rapporteur*
Alexander PRETSCHNER
Professeur à la Technische Universität Kaiserslautern / *Rapporteur*
Mario SÜDHOLT
Maître-assistant à l'École des Mines de Nantes / *Rapporteur*
Jean-Marc JÉZÉQUEL
Professeur à l'Université de Rennes 1 / *Examinateur*
Yves LE TRAON
Professeur à l'Université du Luxembourg / *Examinateur*
Benoît BAUDRY
Chargé de Recherche à l'INRIA / *Examinateur*

Table des matières

Introduction

Pour garantir la qualité des logiciels face à leur complexité croissante, différentes approches existent. Il est possible d'assurer la qualité par construction, en utilisant des techniques qui permettent de modulariser le programme afin de réduire la complexité, comme par exemples les approches orientées objet ou orientées composant. Le test de logiciel vise quant à lui à vérifier la qualité d'un logiciel par des procédés permettant de détecter des fautes.

Afin de maîtriser la complexité des logiciels, l'industrie a massivement adopté la programmation orientée objet, avec des langages tels que Java ou C++. La programmation orientée objet est un paradigme où les données et les méthodes permettant de les manipuler sont encapsulées dans des *objets*. Ce découpage permet de surmonter la complexité d'un programme en augmentant la modularité, en abstrayant les données, et en permettant une réutilisation plus facile de morceaux de programmes.

Un problème, identifié par Kiczales *et al.* [Kiczales 97] persiste cependant : certaines décisions de conception ne peuvent pas être encapsulées dans un objet ou un composant, et sont au contraire dispersées dans différents objets ou composants. Par exemple, le *logging*, qui consiste à enregistrer les événements d'un programme, nécessite de distribuer à travers tout le code du programme des instructions ayant pour objectif l'enregistrement effectif des différents événements. Ceci rend le code plus difficile à lire et à comprendre, augmente le risque d'erreur, et rend la maintenance et l'évolution du code plus difficiles.

Afin de résoudre ce problème, Kiczales *et al.* ont proposé un nouveau paradigme de programmation : *la programmation orientée aspect* (*Aspect-Oriented Programming*). Le principe est de séparer le programme en deux axes orthogonaux. Le programme de base, regroupant les fonctionnalités principales est développé classiquement sur un axe, tandis que les décisions de conception ne pouvant être implémentées de manière unitaire sur l'axe principal sont encapsulées sur l'axe secondaire dans des *aspects*. Un aspect *tisse* un extrait de code, nommé *greffon* (*advice*), à différents endroits du programme.

Ainsi le code est modularisé, ce qui améliore la lisibilité, et facilite l'évolution ou la maintenance. De plus, le code de chaque aspect n'est pas répété, ce qui diminue le risque d'erreur et facilite aussi la maintenance.

Le test de logiciel a pour objectif d'examiner ou d'exécuter un programme dans le but d'y révéler des erreurs. Il est souvent défini comme le moyen par lequel on s'assure qu'un logiciel est conforme à un ensemble d'exigences. Le logiciel est exécuté en utili-

sant différents scénarios de test, puis un oracle vérifie les résultats des exécutions sont conformes aux spécifications.

Dans cette thèse nous défendons qu'il est nécessaire, pour s'assurer de la qualité des logiciels, de combiner les techniques de construction de logiciels complexes avec les techniques de test de logiciel. En effet, il n'est pas envisageable de surmonter la complexité des logiciels au prix de la qualité, ou de n'assurer la qualité que sur des programmes simples.

Dans le cade de cette thèse nous nous sommes donc concentrés sur la combinaison de techniques de test logiciel et de la programmation orientée aspect. De nombreux travaux se sont intéressé au test de programmes orientées objet, mais la programmation orientée aspect étant relativement nouvelle, il reste beaucoup de problème à résoudre pour pouvoir tester des programmes orientés aspect.

Nous défendons que des techniques de test de logiciel efficaces sont nécessaires afin de palier à la lente adoption de la programmation orientée aspect par l'industrie. En 2001, le MIT sélectionnait la programmation orientée aspect parmi les 10 technologies émergentes qui changeront le monde [van der Werff 01]. Malgré cela, la programmation orientée aspect est peu utilisée. Par exemple, dans la communauté libre *Sourceforge*, seuls 0,5 % des projets Java utilisent des aspects.

La première contribution de cette thèse [Munoz 09] vise à **analyser et comprendre quelles peuvent être les limitations pour l'adoption de la programmation orientée aspect**. Le but de cette analyse est de comprendre les raisons de la lente adoption de la programmation orientée aspect et d'identifier les contributions nécessaires pour rendre la programmation orientée aspect plus sûre. Nous nous appuyons sur une étude empirique portant sur 38 projets libres utilisant AspectJ, le langage orienté aspect le plus populaire (au total 479 aspects et une moyenne d'environ 15 000 lignes de code par projet). Nous avons aussi comparé plusieurs implémentations équivalentes d'un même système, en Java et en AspectJ. Ces observations nous ont permis d'analyser comment la programmation orientée aspect est concrètement employée et d'émettre des hypothèses sur les obstacles à son adoption.

Cette étude montre que les développeurs utilisent peu les aspects et que ces aspects sont peu transverses, c'est-à-dire qu'ils sont tissés à peu d'endroits. L'usage des aspects est donc fortement contrôlé, le risque d'effets de bord étant ainsi limité. Nous avons observé que les programmes orientés aspect sont effectivement plus modulaires, comme attendu, mais aussi plus fortement couplés et donc moins testables. Afin d'améliorer la programmation orientée aspect, il est nécessaire de proposer des solutions pour contrôler les effets de bord et pour améliorer la testabilité des programmes orientés aspect, afin d'aider les développeurs à surmonter les difficultés qu'ils peuvent rencontrer et les assister dans le développement de programmes plus modulaires. Les contributions suivantes de la thèse visent donc à fournir des solutions aux problèmes pour le test de programmes orientés aspect que nous avons identifiés grâce à cette étude.

Différents problèmes se posent lorsqu'on teste un programme orienté aspect. Il faut tester le programme en tenant compte des aspects, c'est-à-dire que les oracles des cas de test doivent prendre en compte les modifications apportées par les aspects. Ceci com-

plique la tâche du testeur qui doit comprendre parfaitement la composition qui résulte du tissage des aspects dans le code. Une erreur dans le tissage des aspects peut introduire des erreurs à plusieurs endroits du programme, voire dans tout le programme. L'identification et la localisation de telles fautes peuvent donc s'avérer difficiles. Il est nécessaire de fournir des solutions à ces problèmes.

La seconde contribution de cette thèse [Delamare 08] a pour but de **faciliter la réalisation des tests pour des programmes conçus incrémentalement en utilisant les aspects**. Dans ce cas il est important de pouvoir réutiliser le maximum de cas de test déjà écrits à l'étape de test précédente, tout en modifiant ceux qui sont *impactés* par les aspects, c'est-à-dire ceux qui exécutent les changements introduits par les aspects. Nous proposons une analyse statique de l'impact de l'introduction d'aspects sur les cas de test. Sans avoir recours à l'exécution des suites de tests, l'analyse statique détecte les tests qui ne sont certainement pas impactés par l'introduction d'un aspect. Le testeur peut alors focaliser son effort sur l'adaptation des tests impactés.

La limite principale, à laquelle se heurte toute technique d'analyse statique dans le cas de programmes à objets, est la liaison dynamique et le polymorphisme. Il est impossible de connaître statiquement l'ensemble des méthodes exécutées par un cas de test. De plus, il est parfois nécessaire de sur-approximer le tissage des aspects, car une partie de ce tissage n'est effectuée qu'à l'exécution. Ces deux sur-approximations peuvent entraîner des *faux positifs* – des cas de test non-impactés détectés comme impactés – mais pas de *faux négatifs* – des cas de test impactés non détectés comme impactés. Cette analyse statique a été implémentée dans un outil nommé Vidock[1], qui a servi pour des expérimentations.

Une erreur lors du tissage peut avoir de graves conséquences. Dans de nombreux langages orientés aspect, les points où un greffon est tissé sont décrits à l'aide d'une *expression de point de coupe* (*pointcut descriptor*). Si une expression de point de coupe est incorrecte, un greffon peut être introduit frauduleusement, ou au contraire un greffon peut ne pas être introduit là où il est attendu.

La troisième contribution de cette thèse [Delamare 09a] est une approche pour **le test d'expressions de point de coupe**. Cette approche permet de vérifier la présence ou l'absence d'un greffon à un point de l'exécution sans s'appuyer sur son comportement. Elle s'appuie sur l'outil AdviceTracer, développé au cours de cette thèse, qui supervise l'exécution des greffons au cours du test d'un programme orienté aspect. Contrairement à des cas de test s'appuyant sur le comportement des greffons, les cas de test écrits à l'aide d'AdviceTracer n'échouent que lorsqu'une expression de point de coupe précise n'est pas correcte. La localisation des fautes et l'interprétation des résultats des cas de test sont donc améliorées.

Afin de mesurer les bénéfices de cette approche nous avons mené une étude, où deux développeurs ont écrit une suite de tests sur le même programme AspectJ, l'un utilisant JUnit, l'autre AdviceTracer. Les résultats sur cette étude ont montré que notre approche permet d'écrire des cas de test plus précis, et de détecter des fautes indétectables avec JUnit.

[1] `http://irisa.fr/triskell/Softwares/protos/vidock/`

Enfin, pour que l'ensemble des analyses proposées puissent être exploité, la thèse contient également une contribution technique sous la forme de développements d'outils d'analyse de programmes AspectJ. AjMetrics est un outil mesurant des métriques de couplage sur les programmes AspectJ. La quatrième contribution de cette thèse est **la formalisation d'un modèle abstrait des programmes orientés aspect**. Ce formalisme est une extension de celui proposé par Briand *et al.* [Briand 99] pour les programmes orientés objet. Nous avons défini un ensemble de métriques sur ce formalisme qui peuvent être évaluées sur tout programme orienté aspect, quelque soit le langage de programmation. AjMutator [Delamare 09b] est un outil pour l'analyse de mutation d'expressions de point de coupe en AspectJ. L'analyse de mutation consiste à créer des mutants en insérant des erreurs dans un programme afin d'évaluer la capacité d'une suite de tests à détecter ces erreurs. AjMutator se concentre sur les expressions de point de coupe d'AspectJ, et est capable – dans la majorité des cas – de détecter automatiquement les mutants équivalents, c'est-à-dire qui sont sémantiquement équivalents au programme original. Il s'agit d'une contribution importante, car dans le cas général il s'agit d'un problème indécidable.

Cette thèse est structurée comme expliqué ci-après. Le Chapitre 1 détaille le contexte de cette thèse et établit l'état de l'art. Le Chapitre 2 décrit les études que nous avons menées sur des programmes orientés aspect, et discute les résultats et observations. À partir de ces observations, nous avons identifié plusieurs problèmes pour le test de programmes orientés aspect, pour lesquels nous proposons ensuite des solutions dans les chapitres suivants. D'abord, le Chapitre 3 présente une analyse statique de l'impact des aspects sur les cas de test. Cette analyse permet le test incrémental de programmes orientés aspect en réutilisant le maximum de cas de test possible à chaque ajout d'un aspect. Puis le Chapitre 4 présente une approche pour le test des expressions de point de coupe. Cette approche permet la définition de cas de test visant spécifiquement les erreurs dans les expressions de point de coupe. Ces études et ces différentes solutions ont nécessité des outils d'analyse des programmes orientés aspect, qui sont présentés Chapitre 5. Enfin, la conclusion de cette thèse présente les perspectives pour les travaux futurs.

Chapitre 1

Contexte et état de l'art

Dans ce chapitre nous présentons le contexte de cette thèse, en détaillant à travers l'état de l'art les principaux travaux liés. Dans un premier temps, nous détaillons les principaux concepts de la programmation orientée aspect, puis nous présentons ceux du test de logiciel, et nous terminons par discuter des travaux concernant le test de programmes orientés aspect.

1.1 La programmation orientée aspect

1.1.1 Principes

La programmation orientée aspect est un paradigme de programmation présenté en 1997 par Kiczales *et al.* [Kiczales 97], et dont l'objectif est de séparer les préoccupations transverses des préoccupations principales. Dans les approches de développement traditionnelles, on constate souvent que le code des préoccupations transverses est dispersé dans le code du programme, et ce malgré l'utilisation de patrons de conceptions.

La dispersion du code des préoccupations transverses pose différents problèmes. En multipliant des variations proches d'un même morceau de code, on augmente la probabilité que le développeur introduise une erreur. De plus, il est difficile de faire évoluer le code, car tous les morceaux de code devront être modifiés, et là encore on augmente les probabilités qu'une erreur soit introduite, ou qu'un morceau de code soit oublié et ne soit pas modifié.

Le principe de la programmation orientée aspect est de séparer le code des préoccupations transverses du code de la préoccupation principale, en encapsulant les préoccupations transverses dans des unités appelées «*aspects*». Ainsi le code n'est pas «mélangé» et il devient plus facile de comprendre le code de chaque préoccupation. Le code de chaque préoccupation n'est présent qu'en un seul exemplaire, et sa mise au point ou son évolution s'en trouvent facilités.

La programmation orientée aspect est un paradigme qui peut être utilisé en parallèle avec d'autres paradigmes. Par exemple, il est possible d'utiliser la programmation orientée aspect avec des langages procéduraux, avec des langages orientés objet, des

langages fonctionnels, etc.

Le code de la préoccupation principale et le code des préoccupations transverses sont composés afin de produire le comportement global du programme. Cette composition, appelée «*tissage*» (*weaving*), a le plus souvent lieu à la compilation, mais peut avoir lieu (selon les langages et les outils) à l'exécution. Le tissage est effectué par un outil nommé «*tisseur*»

Lors du tissage, le tisseur insère le code de l'aspect – le «*greffon*» (*advice*) – à certains points du flot d'exécution – nommés «*points de jonction*» (*joinpoints*) –. Ces points de jonction sont désignés par «*l'expression de point de coupe*» (*pointcut descriptor*). Chaque aspect est donc composé d'un greffon et d'une expression de point de coupe.

L'expression de point de coupe

L'expression de point de coupe décrit un ensemble points de jonction où le greffon est tissé. Les points de jonction sont des points dans le flot d'exécution du programme.

Selon les langages, l'expression de point de coupe peut prendre différentes formes et être plus ou moins complexe. Chaque langage orienté aspect définit un langage de point de coupe qui permet de décrire les points de jonction. La nature des points de jonction dépend donc du langage de point de coupe.

Le greffon

Le greffon est le morceau de code qui implémente le comportement de la préoccupation transverse. Lors du tissage, un greffon peut être tissé avant, après, ou autour des points de jonction sélectionnés par l'expression de point de coupe.

Langages orientés aspect

La plupart des langages orientés aspect sont des extensions de langages existants, comme C ou Java. Nous présentons ici quelques un des langes orientés aspect basés sur Java.

AspectJ AspectJ [Kiczales 01] est un des premiers langages orientés aspect, et un des plus utilisés. AspectJ définit un nouveau type de classe, nommé «aspect», qui contient des expressions de point de coupe («pointcut») et des greffons. Une syntaxe spécifique est utilisée pour les expressions de point de coupe et les greffons. Le langage de point de coupe est très expressif. AspectJ est décrit en détail dans la Section 1.1.3.

CaesarJ CaesarJ [Aracic 06] est un langage orienté aspect qui s'appuie sur le langage de point de coupe d'AspectJ et sa syntaxe. CaesarJ introduit un nouveau type de classe, nommé «cclass».

AspectWerkz AspectWerkz [Bonér 04] est un langage orienté objet *léger*, qui n'ajoute pas de nouvelles constructions à Java. Les greffons sont des méthodes java classiques auxquelles sont ajoutées des annotations spécifiant les expressions de point de coupe. Depuis la version 5 d'AspectJ, AspectWerkz a fusionné avec AspectJ.

JBoss AOP Dans JBoss AOP, les greffons sont des méthodes java classiques. Les expressions de point de coupe sont associées aux greffons soit à l'aide d'annotations, soit à l'aide de fichiers de configuration XML. JBoss AOP utilise un langage de point de coupe spécifique proche de celui d'AspectJ.

1.1.2 Exemple introductif

Cette section détaille un exemple particulier de programme orienté aspect, développé en AspectJ. Cet exemple a pour but d'illustrer les différents concepts de la programmation orientée aspect, et sert aussi d'exemple dans les chapitres qui suivent.

L'exemple est un système de ventes aux enchères en ligne, développé dans l'équipe Triskell de l'INRIA Rennes. Un utilisateur peut mettre en vente un objet en précisant une enchère minimum ainsi qu'une date de fin de la vente. D'autres utilisateurs peuvent ensuite enchérir jusqu'à la date de fin où la vente est remportée par le meilleur enchérisseur. Le système s'assure que le vendeur est payé en imposant aux utilisateurs d'avoir assez d'argent libre sur leur compte pour enchérir. Concrètement les utilisateurs possèdent un compte avec dans le système qu'ils peuvent créditer eux-même (avec une carte bancaire par exemple). Ils peuvent débiter de l'argent de leur compte ou l'utiliser pour enchérir s'il est libre. Le montant d'argent libre est le montant disponible sur le compte moins le montant des enchères en cours. Lorsqu'une vente est terminée l'argent est transféré du compte de l'acheteur vers le compte du vendeur.

La Figure 1.1 montre le diagramme de classes du système de ventes aux enchères. Le système contient 44 classes et trois aspects, pour un total de 1 691 lignes de code. La classe `Server` implémente le serveur, et reçoit les requêtes sous la forme de chaînes de caractères via la méthode `query`. Le patron de conception «commande» [Gamma 95] est utilisé pour exécuter les différentes requêtes. La classe `Auction` encapsule une vente. L'état d'une enchère est implémenté à l'aide du patron de conception «état» (`AuctionState`). Les utilisateurs peuvent joindre une enchère et y poster des messages sur un forum (`BulletinBoard`), qui sont modérés par des modérateurs (`Moderator`). Le temps est simulé à l'aide d'entiers représentant une unité de temps abstraite (*tick*). Les dates sont donc exprimées sous la forme d'entiers. Lorsque la méthode `tick` du serveur est appelée, le compteur est incrémenté et le serveur ferme les enchères si leur date de fin est atteinte.

Les trois aspects ajoutés sont les suivants :

`Reserve` Cet aspect offre au vendeur la possibilité de mettre un prix de réserve secret. Si à la fin de la vente le prix de réserve n'est pas atteint, la vente est annulée.

`AltBid` Cet aspect change la façon dont le prix final d'une enchère est calculé. Au lieu que le prix soit le montant de la meilleure enchère, c'est le montant de la

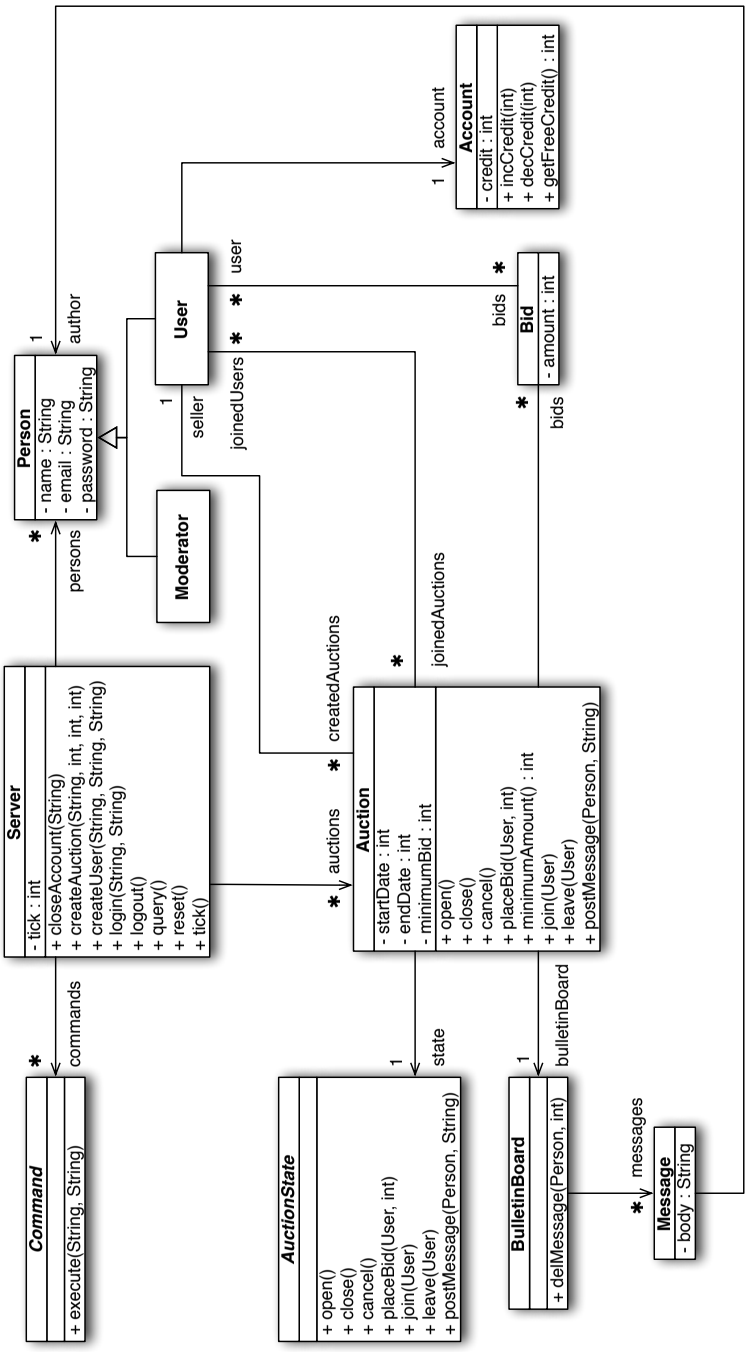

FIG. 1.1 – Diagramme de classes du système de ventes aux enchères.

```
1  public aspect AltBid {
2     pointcut getAmount():
3        call(int Bid.getAmount()) &&
4           cflow(execution(void Open.close(Auction)));
5
6     int around(Bid b): getAmount() && target(b) {
7        /* Le montant de la vente devient le montant de
8          la seconde meilleure enchere plus 10 %. */
9        ...
10    }
11 }
```

Listing 1.1 – Extrait de l'aspect `AltBid`

deuxième meilleure enchère plus 10 %, ou l'enchère minimum s'il n'y a qu'un seul enchérisseur.

Log Cet aspect enregistre toutes les méthodes exécutées dans un fichier.

1.1.3 AspectJ

AspectJ est une extension orientée aspect de Java, créée par PARC (Palo Alto Research Center, anciennement Xerox PARC). Il s'agit d'un des premiers langages orientés aspect, et du plus utilisé. Dans cette thèse, nous nous somme concentrés sur AspectJ.

Afin d'ajouter les concepts de la programmation orientée aspect à Java, AspectJ introduit un type de classe spécialisé, nommé «aspect». Dans un aspect AspectJ, il est possible de définir des champs et méthodes de la même manière que dans une classe Java classique. Il est aussi possible de définir des expressions de point de coupe, des greffons, des définitions inter-types ou des déclarations. Dans cette section nous détaillons ces différents concept.

Exemple

Le Listing 1.1 montre un extrait de l'aspect `AltBid`. Une expression de point de coupe est définie à la ligne 2, `getAmount`. À la ligne 6, un greffon est défini. Le greffon de la ligne 6 est associé à une expression de point de coupe qui référence l'expression de point de coupe `getAmount`.

Langage de point de coupe

En AspectJ, chaque greffon est associé à une expression de point de coupe qui lui est propre. Il est néanmoins possible de définir des expressions de point de coupe qui ne sont pas directement associées à un greffon. Ces expressions de point de coupe sont nommées et peuvent être référencées par une autre expression de point de coupe. Par

exemple dans le Listing 1.1, l'expression de point de coupe `getAmount` (ligne 2) est référencé par l'expression de point de coupe associée au greffon de la ligne 6.

Une expression de point de coupe AspectJ est soit une expression de point de coupe primitive, soit une référence vers une autre expression de point de coupe, soit une composition de différentes expressions de point de coupe. Les expressions de point de coupe primitives permettent de sélectionner des points de jonction et sont la base des expressions de point de coupe.

Types de points de jonction statiques En AspectJ, les points de jonction sont situés dans le flot d'exécution et peuvent donc dépendre de l'état du système à l'exécution. Cependant, à chaque point de jonction correspond un point de jonction statique qui identifie une partie du code source. La partie de code source auquel correspond un point de jonction est aussi appelée «*ombre*» du point de jonction (*joinpoint shadow*).

Les types de points de jonction statiques en AspectJ sont les suivants :

Appel Un point de jonction de type *appel* correspond à l'appel d'une méthode ou d'un constructeur. L'ombre est l'appel en lui-même et peut donc être incluse dans une instruction.

Exécution Un point de jonction de type *exécution* correspond à l'exécution d'une méthode ou d'un constructeur. L'ombre est le corps complet de la méthode ou du constructeur.

Lecture Un point de jonction de type *lecture* correspond à l'accès en lecture à un attribut. L'ombre est l'accès lui-même.

Affectation Un point de jonction de type *affectation* correspond à l'affectation d'un attribut. L'ombre est l'affectation elle-même.

Initialisation statique Un point de jonction de type *initialisation statique* correspond à l'initialisation statique d'une classe, c'est à dire l'initialisation des attributs statiques et l'exécution des blocs d'initialisation statique. L'ombre est le code complet des initialisations statiques de la classe.

Pré-initialisation Un point de jonction de type *pré-initialisation* correspond à la pré-initialisation d'un objet. L'ombre est le code depuis l'entrée dans le premier constructeur jusqu'à l'appel du super-constructeur.

Initialisation Un point de jonction de type *initialisation* correspond à l'initialisation d'un objet. L'ombre est l'appel au premier constructeur.

Gestionnaire d'exception Un point de jonction de type *gestionnaire d'exception* correspond à un bloc de gestion d'exception (de type `catch(..)`). L'ombre est le bloc entier.

Greffon Un point de jonction de type *greffon* correspond à l'exécution d'un greffon. L'ombre est le code complet du greffon.

Motifs Afin de désigner des éléments du programmes dans les expressions de point de coupe, il est nécessaire d'utiliser des «*motifs*» (*pattern*). Il existe quatre types de motif dans le langage de point de jonction d'AspectJ : les motifs de classe, les motifs de méthode, les motifs de constructeur, et les motifs d'attribut. Chaque motif possède une syntaxe propre qui permet de désigner un ou plusieurs éléments.

Afin de désigner plusieurs éléments, il est possible d'utiliser des *jokers* (*wildcard*). Il existe deux types de jokers. Le premier, «`*`» désigne n'importe quelle suite de caractères, et peut être utilisée dès qu'un nom est attendu (nom de méthode, de classe, etc.). Le second, «`..`» désigne n'importe quelle suite de paramètres, et peut être utilisée dès qu'un ou plusieurs paramètres sont attendus (paramètres d'une méthode ou d'un constructeur).

Par exemple, le motif de méthode «`int Bid.getAmount()`» (Listing 1.1, ligne 3) sélectionne la méthode sans paramètre `getAmount` de la classe `Bid`. Le motif de méthode «`* Bid.get*()`» sélectionne toutes les méthodes sans paramètre de la classe `Bid` dont le nom commence par «get». Le motif «`* *.*(..)`» sélectionne toutes les méthodes du système, quelque soit le type de retour, la classe, le nom de la méthode, ou les paramètres.

Expressions de point de coupe primitives Les expressions de point de coupe primitives sont au nombre de 17, et sont réparties en trois catégories. La première catégorie sélectionne des points de jonction statiques. La seconde catégorie restreint statiquement l'ensemble des points de jonction sélectionnés, tandis que la dernière catégorie restreint dynamiquement l'ensemble des points de jonction sélectionnés.

Sélection de points de jonction statiques Ces expressions de point de coupe primitives sélectionnent des points de jonction statiques :

`call`(*Motif de méthode ou de constructeur*) Sélectionne les appels aux méthodes décrites par le motif de méthode (points de jonction statiques de type *appel*).

`execution`(*Motif de méthode ou de constructeur*) Sélectionne l'exécution des méthodes décrites par le motif de méthode (points de jonction statiques de type *exécution*).

`get`(*Motif d'attribut*) Sélectionne les accès en lecture des attributs décrits par le motif d'attribut (points de jonction statiques de type *lecture*).

`set`(*Motif d'attribut*) Sélectionne les affectations des attributs décrits par le motif d'attribut (points de jonction statiques de type *affectation*).

`staticinitialization`(*Motif de classe*) Sélectionne l'initialisation statique des classes décrites par le motif de classe (points de jonction statiques de type *initialisation statique*).

`preinitialization`(*Motif de constructeur*) Sélectionne les pré-initialisations des constructeurs décrits par le motif de constructeur (points de jonction statiques de type *pré-initialisation*).

`initialization`(*Motif de constructeur*) Sélectionne les initialisations effectuées par les constructeurs décrits par le motif de constructeur (points de jonction statiques de type *initialisation*).

`handler`(*Motif de type*) Sélectionne les blocs de gestions d'exception pour les exceptions décrites par le motif de type (points de jonction statiques de type *gestionnaire d'exception*).

`adviceexecution` Sélectionne toutes les exécutions de greffons (points de jonction de type *greffon*). Cette expressions de point de coupe primitive ne prend pas de paramètre.

Contraintes statiques Les expressions de point de coupe primitives suivantes restreignent statiquement les points de jonctions sélectionnés, c'est-à-dire que les contraintes peuvent être résolues à la compilation.

`within`(*Motif de type*) Restreint aux points de jonction dont l'ombre se trouve dans le code des classes décrites par le motif de type.

`withincode`(*Motif de méthode ou de constructeur*) Restreint aux points de jonction dont l'ombre se trouve dans le code des méthodes décrites par le motif de méthode.

Contraintes dynamiques Les expressions de point de coupe primitives suivantes restreignent dynamiquement les points de jonction sélectionnés, c'est-à-dire que les contraintes ne peuvent être résolues qu'à l'exécution. Ces expressions de point de coupe ne peuvent pas utiliser de *joker* car AspectJ ne peut résoudre les *joker* qu'à la compilation.

`this`(*Classe*) Restreint aux points de jonction où l'objet courant est une instance de la classe passée en paramètre.

`target`(*Classe*) Restreint aux points de jonction où l'objet cible est une instance de la classe passée en paramètre.

`args`(*Classes*) Restreint aux points de jonction dont les paramètres sont des instances des classes passées en paramètre.

`cflow`(*Expression de point de coupe*) Restreint aux points de jonction se trouvant dans le flot de contrôle d'au moins un des points de jonction sélectionnés par l'expression de point de coupe passée en paramètre.

`cflowbelow`(*Expression de point de coupe*) Idem que `cflow`, excepté que les points de jonction sélectionnée par l'expression de point de coupe passée en paramètre ne peuvent pas être sélectionnés.

`if`(*Expression booléenne Java*) Restreint aux points de jonction qui satisfont l'expression booléenne Java passée en paramètre.

Expressions de point de coupe statiques et expressions de point de coupe dynamiques Une expression de point de coupe statique est une expression de point de coupe qui peut-être calculée statiquement car elle ne contient pas de expression de point de coupe primitive dynamique. Au contraire une expression de point de coupe dynamique ne peut être calculée qu'à l'exécution car elle contient une contrainte dynamique.

À la compilation, seule la partie statique d'une expression de point de coupe dynamique est calculée. Lors de l'exécution les contraintes dynamiques sont résolues afin de déterminer si le greffon doit être exécuté ou pas.

Composition des expressions de point de coupe Pour construire des expressions de point de coupe complexes, il est possible de composer différentes expressions de point de coupe. L'opérateur binaire de conjonction («`&&`») restreint les points de jonction sélectionnés aux points de jonction sélectionnés par les deux expressions de point de coupe. L'opérateur binaire de disjonction («`||`») permet de sélectionner les points de jonction sélectionnés par une des deux expressions de point de coupe. L'opérateur unaire de négation (« `!`») sélectionne les points de jonction qui ne sont pas sélectionnés par l'expression de point de coupe.

Exemple

Le Listing 1.1 montre un extrait de l'aspect `AltBid`. À la ligne 2, l'expression de point de coupe `getAmount` est définie. Cette expression de point de coupe sélectionne tous les appels à la méthode `Bid.getAmount` qui sont dans le flot de contrôle de l'exécution de `Open.close`. Lors de l'exécution, à chaque appel de la méthode `Bid.getAmount`, on vérifie s'il a lieu dans le flot de contrôle de l'exécution de la méthode `Open.close`. Si c'est le cas alors le greffon est exécuté, sinon l'exécution continue normalement.

Greffon

Les greffons AspectJ sont écrits en Java, et leur corps diffère très peu du corps d'une méthode Java traditionnelle. À chaque greffon est associé une expression de point de coupe qui désignes les points de jonction où il est tissé.

Un greffon peut être tissé *avant* (mot-clé «`before`»), *après* («`after`») ou *autour* («`around`») de l'ombre des points de jonction sélectionnés par son expression de point de coupe.

Lorsqu'un greffon est tissé avant ou après les points de jonction, l'ombre des points de jonction est toujours exécutée (sauf dans le cas particulier où un greffon s'exécute avant et lève une exception). Un greffon tissé avant ou après n'a pas de type de retour.

Lorsqu'un greffon est tissé autour des points de jonction, l'ombre des points de jonction n'est pas exécutées, et le greffon est exécuté à la place. En appelant une fonction spécifique, «`proceed`», le greffon a la possibilité d'exécuter l'ombre. La fonction

«`proceed`» peut être exécutée plusieurs fois (l'ombre est exécutée à chaque fois), ou peut ne pas être appelée (auquel cas l'ombre n'est jamais exécutée).

Un greffon tissé autour doit retourner une valeur, qui doit être de même type que celle des points de jonction autour desquels il est tissé. Par exemple le greffon du Listing 1.1, ligne 6, est tissé autour des appels à la méthode `Bid.getAmount()`. Cette méthode retourne un entier, et donc le greffon retourne aussi un entier. Ceci permet dans ce cas de modifier la valeur retournée par l'appel.

Définitions inter-types

Les définitions inter-types sont des définitions qui ont lieu dans des aspects AspectJ, et qui ajoutent des méthodes ou des attributs à des classes Java existantes. Les définitions inter-types ne peuvent pas utiliser les *jokers*, et chaque définition est donc spécifique à une classe.

La visibilité des définitions inter-types sont du point de vue de l'aspect déclarant. Ainsi un membre inter-types *privé* n'est visible que par l'aspect qui le définit. Il n'est pas possible de définir un membre inter-types comme *protégé*. Par défaut un membre inter-type est visible par tout le paquet où se trouve l'aspect déclarant.

Déclarations

En plus des greffons et des définitions inter-types, AspectJ introduit quatre types de déclarations. Ces déclarations permettent de spécifier la priorité des aspects, *d'adoucir* les exceptions, de modifier la hiérarchie des classes, ou de lever des avertissements ou erreurs.

Dans le cas où deux greffons sont tissés au même point de jonction, il est nécessaire de préciser la priorité des aspects à l'aide de déclarations spécifiques. Il est possible d'utiliser des *jokers*, par exemple pour spécifier qu'un aspect est prioritaire sur tous les autres.

Un aspect peut spécifier qu'un type d'exception ne doit pas obligatoirement être déclaré. En Java une méthode doit déclarer les exceptions qu'elle est susceptible de lever, AspectJ permet de contourner cette obligation.

Un aspect peut modifier la hiérarchie des classes en ajoutant un parent à une classe. Il est toujours possible d'ajouter des interfaces comme parent, mais il faut s'assurer que les méthodes de l'interface sont implémentées (par exemple en utilisant des définitions inter-types). Il n'est possible d'ajouter une classe comme parent que si la classe fille n'a pas d'autre ancêtre que la classe `Object`.

AspectJ permet aussi de spécifier de nouveaux avertissements ou erreurs. Ce type de déclaration est associé à une expression de point de coupe statique. Une erreur ou un avertissement sera retourné par le compilateur pour chaque point de jonction sélectionné par l'expression de point de coupe.

1.1.4 Problèmes et critiques de la programmation orientée aspect

Si la programmation orientée aspect résout des problèmes dûs à la dispersion du code des préoccupations transverses, elle introduit aussi différents problèmes et concentre certaines critiques. Nous discutons ici de ces problèmes et critiques, ainsi que des solutions ayant été apportées.

Fragilité de l'expression de point de coupe

La *fragilité de l'expression de point de coupe* (aussi appelé *paradoxe de l'évolution*) est un problème défini par Kellens *et al.* [Kellens 06] comme suit :

> «Le problème de la fragilité de l'expression de point de coupe se produit quand une expression de point de coupe capture involontairement ou rate un point de jonction particulier du fait de sa fragilité à l'égard de modifications apparemment sans danger du programme de base.»

Il n'est donc pas possible de déterminer si une modification est sûre en n'examinant que le programme de base.

Par exemple, l'expression de point de coupe `getAmount` du Listing 1.1 désigne explicitement les méthodes qu'elle référence (`Bid.getAmount` et `Open.close`). Si, lors d'une évolution du code, le nom d'une de ces méthodes change, l'expression de point de coupe n'est plus correcte, et le greffon n'est pas tissé correctement, ce qui peut entraîner des dysfonctionnements graves.

Lors de l'évolution d'un programme orienté aspect, la fragilité de l'expression de point de coupe peut entraîner trois types d'erreurs différentes :

1. un point de jonction où était tissé un greffon n'est plus sélectionné par l'expression de point de coupe
2. un nouveau point de jonction est incorrectement sélectionné par une expression de point de coupe
3. un nouveau point de jonction n'est pas sélectionné par une expression de point de coupe alors que le greffon devrait y être tissé

Plusieurs solutions ont été proposées pour le problème de la fragilité de l'expression de point de coupe, comme discuté par Munoz *et al.* [Munoz 07]. Ces solutions peuvent être réparties en quatre catégories.

Solutions basées sur les langages Ces solutions proposent de nouveaux langages ou des extensions pour des langages existants. L'objectif est de limiter le couplage entre les expressions de point de coupe et le code source.

Kiczales et Mezini [Kiczales 05] proposent les fondations des *points de jonction explicites*. Le principe des points de jonction explicites est que le programme de base et les aspects sont composés à l'aide d'interfaces qui définissent explicitement où sont tissés les aspects. Le programme de base et les aspects sont donc découplés, et les interfaces sont stables lors de l'évolution. Les aspects et le code de base peuvent donc

évoluer individuellement tant qu'ils utilisent correctement les interfaces. La contrepartie est que cela ne respecte pas le principe *d'obliviousness*, qui veut que le programme de base n'ait pas connaissance des aspects. Hoffman et Eugster [Hoffman 08] ont proposé une approche similaire et ont mené une étude empirique sur différents programmes, en comparant à chaque fois trois implémentations, une avec Java, une avec AspectJ, et une avec les points de jonction explicites. Cette étude montre que les aspects utilisant les points de jonction explicites peuvent mieux être réutilisés et que les expressions de point de coupe sont plus simples, mais aussi que la modularité est réduite.

De Fraine *et al.* [De Fraine 08] ont proposé un langage basé sur AspectJ, StrongAspectJ, dont le but est de rendre plus sûr les interactions entre les greffons et le programme de base, en renforçant le typage des greffons et des points de jonction. Les auteurs identifient différents problèmes dus au système de typage utilisé par AspectJ, en particulier dans le cas de greffons tissés autour d'un point de jonction (mot-clé `around`). La méthode `proceed`, qui permet d'exécuter le point de jonction où est tissé le greffon, a en AspectJ la même signature que le greffon, ce qui peut provoquer des incompatibilités de types dans certains cas. Les auteurs proposent une solution basée sur un langage, StrongAspectJ, qui permet de définir une signature différente pour la méthode `proceed`. StrongAspectJ permet aussi de typer les greffons de manière générique, pour plus de flexibilité.

Pawlak *et al.* [Pawlak 04] ont proposé un *framework* pour Java qui utilisent des composants réutilisables pour encapsuler les greffons. Ainsi les greffons ne sont pas directement liés à des expressions de point de coupe. Des composants nommés *wrappers* encapsulent les greffons, et des composants nommés *aspect-components* lie dynamiquement les greffons aux composants de base du système. Ainsi les notions de greffon et d'expression de point de coupe sont découplés pour une meilleure modularité.

D'autres solutions basées sur les langages ont été proposées par Gybels et Brichau [Gybels 03], Sakurai et Masuhara [Sakurai 08], Griwold *et al.* [Griswold 06], Aldrich [Aldrich 05], ou Ostermann *et al.* [Ostermann 05]

Analyse Ces solutions sont basées sur une analyse des propriétés et du comportement du programme.

Koppen et Stoerzer [Koppen 04], ainsi que Stoerzer et Graf [Stoerzer 05], ont proposé une analyse pour le problème de la fragilité de l'expression de point de coupe. Cette approche, nommée *analyse delta*, consiste à comparer, lors d'une évolution, l'ensemble des points de jonction sélectionnés par chaque expression de point de coupe afin de produire un ensemble *delta* contenant les points de jonction retirés et les points de jonction ajoutés. Cet ensemble peut ensuite servir à vérifier qu'une expression de point de coupe est correcte, ou à localiser une faute découverte par ailleurs. Cette approche est supportée par un outil, PCDiff, une extension pour l'IDE Eclipse. PCDiff est implémenté pour AspectJ, mais l'approche fonctionne pour n'importe quelle langage où les expressions de point de coupe sont calculables statiquement (au moins partiellement).

Lopez-Herrejon *et al.* [Lopez-Herrejon 06] ont aussi proposé une analyse pour identifier des problèmes liés à la composition des aspects.

Solutions basées sur les modèles Ces solutions utilisent des abstractions haut niveau pour éviter les problèmes liés à la fragilité de l'expression de point de coupe.

Kellens *et al.* [Kellens 06] proposent d'utiliser des expressions de point de coupe basées sur des modèles. Ces expressions de point de coupe sont définies sur un modèle conceptuel du programme plutôt que de faire directement référence à l'implémentation du programme. Par exemple, le modèle conceptuel peut contenir un ensemble d'accesseurs, ce qui évite de définir un motif de méthode de type «`* get*()`». Ce motif est basé sur une convention de nommage des méthodes, or selon cette convention, un accesseur pour un booléen commence par «`is`» plutôt que par «`get`». Ce motif ne permet donc pas de capturer un accesseur qui serait ajouté pour un booléen, ou simplement un accesseur qui n'utiliserait pas cette convention de nommage. Une expression de point de coupe référençant les accessseurs du modèle conceptuel sera toujours correcte si le modèle est mis à jour lors de l'évolution.

Contrats pour les aspects Ces solutions utilisent la programmation par contrat [Meyer 92] pour repérer d'éventuels problèmes liés à la fragilité de l'expression de point de coupe.

Skotiniotis et Lorenz [Skotiniotis 04] proposent des notions de programmation par contrat pour les aspects. Ces contrats s'assurent que les aspects sont tissés correctement et n'interfèrent pas avec d'autres parties du programme.

D'autres solutions pour la programmation par contrat ont été proposées pour la programmation orientée aspect par Klaeren *et al.* [Klaeren 01], Lagaisse *et al.* [Lagaisse 04], et Munoz *et al.* [Munoz 08].

Critiques

Plusieurs critiques ont été émises concernant la programmation orientée objets. Elles pointent les limites de la programmation orientée aspect qui doivent être repoussées par de nouvelles solutions ou des nouveaux langages orientés aspect.

Une critique émise par Beuche et Beust [Beuche 06] et par Steimann [Steimann 06] est qu'au-delà de quelques exemples précis et connus – *profiling*, historique d'exécution, sécurité etc. – il n'y a pas ou peu de préoccupations transverses. Il est aussi reproché que les préoccupations classiques sont avant tout des problèmes techniques rencontrés par les développeurs. Il ne serait donc pas nécessaire d'utiliser un nouveau paradigme pour quelques cas précis, et l'utilisation de pratiques de programmation, tels les patrons de conception, pourrait donc suffire.

Certaines critiques reprochent à la programmation orientée aspect d'implémenter *l'anti-patron action à distance* (*action at a distance*). Un anti-patron correspond à de mauvaises pratiques de programmation et sert de contre-exemple. Dans l'anti-patron *action à distance*, le comportement d'une classe varie selon des opérations dans une autre classe, qui sont difficiles voire impossibles à identifier. Il devient donc difficile à comprendre les interactions avec les autres modules du programme, contrairement par exemple à une classe dans la programmation orientée objet où les interactions ont lieu

via une interface définie. Beuche et Beust [Beuche 06] précisent notamment qu'il est difficile d'expliquer le code à un développeur qui ne le connaît pas.

Pour Steimann [Steimann 06], le succès de la programmation orientée aspect est paradoxale. Il défend que, contrairement à l'objectif annoncé, la programmation orientée aspect réduit la modularité. Pour l'auteur, un des avantages de la programmation orientée aspect est le regroupement du code d'une préoccupation au même endroit, mais cela sacrifie le principe de localité spatiale [Denning 05] (les instructions et données utilisées sont proches des instructions et données utilisées précédemment). Cette critique rejoint notamment les critiques sur l'anti-patron *action à distance*.

1.1.5 Classifications des Aspects

Plusieurs travaux ont proposé des classifications des aspects. Ces classifications utilisent différents critères, et permettent aux développeurs une meilleure compréhension du programme et une identification plus aisée d'éventuels problèmes. Dans le cadre de cette thèse, nous avons utilisé une classification des aspects, proposée par Munoz *et al.* [Munoz 08], lors d'une étude empirique, afin de mieux comprendre l'utilisation des aspects par les développeurs.

Rinard *et al.* [Rinard 04] ont proposé une classification des greffons basée sur les interactions entre les aspects et le programme de base. Les greffons sont classés selon deux axes ; le premier axe correspond aux interactions directes entre un greffon et les méthodes où il est tissé, tandis que le second correspond aux interactions indirectes entre un greffon et des méthodes qui accéderaient aux mêmes attributs.

La classification pour les interactions directes est la suivante :

Augmentation Le corps des méthodes où est tissé le greffon s'exécute toujours entièrement.

Rétrécissement Soit le corps des méthodes où est tissé le greffon s'exécute entièrement, soit il ne s'exécute pas du tout. Les méthodes sont en fait exécutées sous une certaine condition.

Remplacement Le corps des méthodes où est tissé le greffon ne s'exécute pas du tout. Le greffon remplace donc le comportement de ces méthodes.

Combinaison Le corps du greffon et des méthodes où il est tissé se combinent pour former un nouveau comportement.

La classification pour les interactions indirectes est la suivante :

Orthogonal Le greffon et les méthodes du programme de base accèdent à des attributs différents, il n'y a donc pas d'interaction indirecte.

Indépendant Le greffon n'écrit pas dans des attributs auxquels les méthodes du programme de base peuvent accéder, et les méthodes du programme de base n'écrivent pas dans des attributs auxquels le greffon peut accéder.

Observation Le greffon peut lire des attributs auxquels accèdent les méthodes de base, le reste des interactions étant similaires à celles d'un greffon indépendant.

Action Le greffon peut lire et écrire des attributs auxquels accèdent les méthodes de base, le reste des interactions étant similaires à celles d'un greffon indépendant.

Interférence Le greffon et les méthodes accèdent en lecture et écriture aux mêmes attributs.

Ces deux axes de classification sont orthogonaux, et dans chaque axe les classifications sont mutuellement exclusives. Par exemple un greffon qui observe le programme de base et enregistre ces informations dans un fichier sera classé comme une *augmentation* pour le premier axe, et comme *indépendant* pour le second axe.

Les auteurs proposent une analyse statique de programmes AspectJ qui permet de classer automatiquement les greffons. L'objectif est d'aider le développeur à repérer d'éventuels problèmes dans les interactions entre les greffons et le programme de base. Si par exemple un greffon sensé *observer* le programme de base est classé comme *interférant*, il y a une erreur dans le greffon ou dans le tissage avec le programme de base.

Munoz *et al.* [Munoz 08] ont proposé une classification des aspects et des greffons se basant sur leur caractère *invasif*. Un aspect est considéré comme invasif s'il manipule le flot de données ou le flot de contrôle du programme de base. Les auteurs proposent une liste de structures invasives ; ces structures peuvent être implémentées par un aspect ou par un greffon.

Les structures invasives pouvant être implémentées par des aspects sont les suivantes :

Hiérarchie L'aspect modifie la hiérarchie des classes, par exemple en ajoutant un parent à une classe existante.

Ajout d'un attribut L'aspect ajoute un nouvel attribut à une classe existante.

Ajout d'une méthode L'aspect ajoute une nouvelle méthode à une classe existante.

Les structures invasives pouvant être implémentées par des greffons sont les suivantes :

Augmentation Le greffon augmente le comportement du programme de base, mais ne modifie pas le comportement existant.

Remplacement Le greffon remplace totalement un comportement existant par un nouveau.

Remplacement conditionnel Le greffon remplace sous une certaine condition un comportement existant par un nouveau. Selon cette condition soit le nouveau comportement est exécuté, soit l'existant.

Multiple Le comportement du point de jonction où est tissé le greffon est exécuté plusieurs fois, ce qui peut donc induire un nouveau comportement.

Croisement Le greffon invoque une ou plusieurs méthodes du programme de base. Le greffon a donc une dépendance vers les classes dont il invoque les méthodes.

Écriture Le greffon affecte des valeurs à un ou plusieurs attributs du programme de base.

Lecture Le greffon lit les valeurs de un ou plusieurs attributs du programme de base.

Argument Lors de l'invocation d'une méthode, le greffon modifie un ou plusieurs des arguments de cette méthode.

Ces différentes structures invasives ne sont pas forcément exclusives. Par exemple un greffon peut implémenter les structures *lecture* et *écriture*. Un aspect ou un greffon qui implémente une structure invasive est dit *invasif*.

Les auteurs utilisent cette classification dans un outil, ABIS, qui permet de spécifier dans du programme de base quels sont les classes d'aspects autorisées à être tissées à un endroit donné. Les aspects et greffons sont automatiquement classés lors de la compilation – l'analyse est donc statique – et si la classe d'un aspect ou d'un greffon est incompatible avec les endroits où il est tissé, un avertissement est remonté au développeur.

Pour les études empiriques présentées dans le Chapitre 2, nous avons utilisé ABIS, et la classification des aspects invasifs pour étudier précisément l'utilisation des aspects invasifs par les développeurs.

1.2 Test de logiciel

Les erreurs dans les logiciels sont courantes et très coûteuses. En 2002 le *National Institute of Standards and Technology* [NIST 02] (NIST, une agence du Département du Commerce des États-Unis) a estimé que les erreurs dans les logiciels coûtaient chaque année 59,5 milliards de dollars à l'économie des États-Unis :

> «L'impact des erreurs logicielles est énorme car virtuellement toute l'économie des États-Unis repose maintenant sur des logiciels pour le développement, la production, la distribution, et le service après-vente des produits et services.»

Le NIST estime que plus d'un tiers de ce coût peut être évité en identifiant plus tôt les erreurs.

Il est donc nécessaire de s'assurer qu'un logiciel est correct, est ceci tout au long de son cycle de vie. Le test de logiciel est généralement la solution adoptée pour s'assurer qu'un logiciel est correct.

1.2.1 Principes du test de logiciel

Le test logiciel [Myers 79, Beizer 90, Binder 99, Ammann 08] consiste à vérifier *empiriquement* la qualité d'un logiciel. La vérification est partielle car le test d'un logiciel ne peut pas être exhaustif.

Le test de logiciel peut intervenir à différents étapes du développement d'un logiciel, selon la technique utilisée. Le test unitaire vérifie que chaque unité du programme a été correctement implémentée, le test d'intégration vérifie que les interactions entre les différentes unités sont correctes, et le test système vérifie les fonctionnalités du système complet.

Le but du test unitaire est de s'assurer que chaque unité du programme fonctionne correctement, indépendamment des autres unités. L'unité du programme dépend du paradigme de programmation utilisé. Par exemple dans la programmation procédurale l'unité est la procédure, tandis que dans la programmation orientée objet l'unité est la classe.

Le test de logiciel peut être statique ou dynamique. Dans le cas du test statique, le code est analysée manuellement ou automatiquement afin de détecter des erreurs. Dans le cas du test dynamique, le programme est exécutée afin de vérifier son comportement. Dans le cas de cette thèse nous n'avons considéré que le test dynamique.

Test dynamique de logiciel

Le test dynamique consiste à exécuter une *suite de tests* afin de détecter d'éventuelles erreurs. Une suite de tests est composée de différents *cas de test*. Lorsqu'une suite de tests est exécutée chaque cas de test est exécuté successivement.

Un cas de test est composé d'un *scénario de test*, d'une *donnée de test*, et d'un *oracle*. Un cas de test correspond à une intention, c'est-à-dire qu'il cherche à vérifier une partie spécifique du programme, ou une propriété, etc. Cette intention est traduite par une *donnée de test* et un *scénario de test*. La donnée de test est la donnée d'entrée du programme. Le scénario de test décrit l'état du système et le déroulement du cas de test. *L'oracle* vérifie que l'exécution du cas de test s'est déroulée correctement, d'après la spécification du programme.

Le test dynamique peut-être fonctionnel ou structurel. Dans le cas du test fonctionnel le programme est considéré comme une «*boîte noire*» et seule les entrées et sorties du programme sont considérées. Dans le cas du test structurel le programme est considéré comme une «*boîte blanche*», et les cas de test peuvent cibler certaines parties du programme.

Verdict d'un cas de test

Le verdict d'un cas de test est émis par l'oracle. L'oracle peut être manuel ou automatisé. Lorsqu'il est automatisé il existe plusieurs techniques pour définir un programme.

Avec un oracle manuel, le testeur doit vérifier lui-même que le résultat de chaque cas de test est correct. Ce genre d'oracle est souvent utilisé sur des programmes de très petite taille, mais il est souvent nécessaire d'automatiser l'oracle afin que le test soit moins coûteux.

L'oracle peut être la sortie attendue du programme. Dans ce cas, un cas de test n'est composé que d'une donnée de test et d'un résultat attendu. Il suffit donc de comparer la sortie obtenue avec la sortie attendue. Ce genre d'oracle est simple et efficace mais n'est pas toujours possible. Il n'est parfois pas possible de calculer la sortie attendue, ou alors cela est possible mais trop coûteux.

Dans le cas du test dirigé par les modèles (*model-based testing*) [Pretschner 05, Utting 06], l'oracle est le modèle comportemental du programme, qui

est comparé au comportement effectif du programme. Le comportement du programme est décrit par un modèle (diagrammes d'états, réseaux de Petri, etc.), sur lequel on sélectionne une trace. Cette trace contient les données de test ainsi que le résultat attendu, elle constitue donc un cas de test complet avec l'oracle.

La solution généralement utilisée est l'oracle partiel. Un oracle partiel est composé d'assertions qui vérifient un certain nombre de propriétés. Si toutes ces propriétés sont vraies, le cas de test *réussit*, sinon il *échoue*. L'oracle est dit partiel car les assertions ne sont pas exhaustives et ne vérifient pas l'intégralité du résultat.

Critère minimum de test

Le test de logiciel, et en particulier le test dynamique, ne peut être exhaustif car l'ensemble des données d'entrée est soit non borné soit trop grand. Il est donc souvent nécessaire de savoir quand une suite de tests est *suffisante*.

Un critère minimum de test est une propriété qui permet d'évaluer si une suite de tests est *suffisante* en définissant un objectif à atteindre pour la suite de tests. La satisfaction de différents critères minimum n'est pas une garantie qu'une suite de tests détectera toutes les erreurs.

Un critère de test peut être basé sur la structure du programme dans le cas du test structurel. Par exemple le critère peut-être basé sur la couverture du code, du flot de contrôle ou du flot de données [Harrold 94]. Dans le cas du test dirigé par les modèles, il est possible de définir des critères de couverture des modèles comportementaux [Pretschner 05].

Un critère de test peut être basé sur la spécification du programme dans le cas du test fonctionnel. Par exemple le critère peut être basé sur une partition du domaine d'entrée du programme [Ostrand 88].

1.2.2 Test de non-régression

Le test de non-régression est un type de test dont l'objectif est de s'assurer qu'il n'y a pas de *régression* lors d'une évolution. Une régression a lieu lorsqu'un fonctionnalités qui fonctionnaient avant l'évolution ne fonctionne plus ensuite. Le test de non-régression doit être effectué à différents niveaux de test (test unitaire, test d'intégration, etc.).

Afin d'éviter de recommencer toute la phase de test à chaque évolution d'un programme, il est nécessaire d'utiliser les cas de test de la version précédente. Le problème est que certains de ces cas de test doivent être modifié ou supprimés pour prendre en compte les changements apportés lors de l'évolution du programme. L'identification de ces cas de test peut être très coûteuse et il est nécessaire de fournir des solutions adaptées au test de non-régression.

Différentes solutions de sélection de cas de test pour le test de non-régression ont été proposées [Rothermel 96]. Harrold *et al.* [Harrold 01] ont proposé une analyse statique de programmes Java pour le test de non-régression. Cette solution, basée sur des travaux de Rothermel et Harrold [Rothermel 97], consiste à construire un graphe de

flot de contrôle de la version précédente et de la nouvelle version du programme. Pour chaque cas de test il faut aussi connaître la couverture du graphe de flot de contrôle de la version précédente. En comparant les différents graphes et la couverture de chaque cas de test, il est possible de déterminer quels cas de test ne couvre aucun changement et peuvent donc être réutilisés et quels cas de doivent être modifiés.

Dans le Chapitre 3 nous effectuons une analyse statique permettant d'identifier les cas de test qui exécutent un aspect nouvellement introduit. Cet analyse est proche des analyses pour le test de non-régression, comme discuté dans la Section 1.3.

1.2.3 Testabilité et métriques logicielles

La testabilité est une propriété qualifiant l'effort nécessaire pour tester un programme ou une partie d'un programme. Pour évaluer la testabilité d'un logiciel, il est nécessaire d'utiliser des métriques logicielles [Bache 90]. Bruntik et van Deursen [Bruntink 06] ont étudié la corrélation entre la testabilité et différentes métriques concernant le code source de programmes orientés objet.

Une métrique est une valeur quantitative servant à évaluer une propriété qualitative. Les métriques logicielles sont mesurées sur les programmes et peuvent porter sur la taille des programmes, les interactions entre différents modules, etc. L'intérêt de métriques de testabilité est de pouvoir prédire la testabilité d'un logiciel, ou d'identifier les techniques qui permettent d'améliorer la testabilité.

Briand *et al.* [Briand 99] ont proposé un formalisme pour unifier la définition de métriques sur les programmes orientés objet. Ce formalisme abstrait les concepts génériques de la programmation orientée objet afin de pouvoir s'adapter à la majorité des langages. Les métriques étant définies à l'aide du formalisme, elles n'ont pas besoin d'être redéfinies pour chaque langage. Les métriques définies sur le formalisme de Briand *et al.* ont été proposées par Li et Henry [Li 93], Chidamber et Kemerer [Chidamber 94], Churcher et Shepperd [Churcher 95], et Henderson-Sellers [Henderson-Sellers 96]

Dans le Chapitre 2, nous avons mesuré des métriques logicielles sur des programmes orientés aspect et des programmes orientés objet. Pour cela nous avons étendu le formalisme de Briand *et al.*, et nous avons développer un outil, AjMutator, implémentant ce formalisme. AjMutator et l'extension au formalisme sont détaillés dans le Chapitre 5.

1.2.4 Analyse de mutation

L'*analyse de mutation* est une technique fondée sur l'injection systématiques de fautes et qui vise à qualifier la capacité d'une suite de tests à détecter des erreurs dans un programme donné.

L'analyse de mutation a été proposée par DeMillo *et al.* [DeMillo 78]. Des *opérateurs de mutation* produisent des *mutants*. Un mutant est une copie du programme dans laquelle une erreur a été injectée. Chaque opérateur de mutation injecte un type de faute précis.

Le *score de mutation* évalue la capacité d'une suite de tests précise à détecter des fautes dans le programme original. La suite de tests est exécutée successivement sur tous les mutants. Chaque mutant est soit *tué* par la suite de tests soit il est *vivant*. Le score de mutation est le pourcentage de mutants tués.

L'analyse de mutation a d'abord pour but de qualifier les données de test. Dans ce cas on considère qu'un mutant est *tué* si une suite de tests est capable d'exhiber une différence entre le programme original et le programme mutant. Dans le cas contraire le mutant est *vivant*.

Il est aussi possible de qualifier les cas de test en entier, c'est-à-dire les données de test et l'oracle. On considère alors qu'un mutant est *tué* si au moins un cas de test a un verdict différent sur le programme original et sur le programme mutant. Dans ce cas, pour qu'un mutant soit tué il faut à la fois que la donnée de test puisse exhiber une différence, mais qu'en plus l'oracle soit capable d'observer cette différence.

L'analyse de mutation est fondée sur deux hypothèses, le *programmeur compétent* et *l'effet de couplage*. L'hypothèse du programmeur compétent suppose que le programme n'est peut être pas correct, mais qu'il est proche du programme correct, les modifications pour le corriger sont donc simples. L'hypothèse de l'effet de couplage suppose que si une suite de tests peut détecter les erreurs simples, elle peut détecter les erreurs complexes. Offut [Offutt 92] a mené des expériences sur l'effet de couplage qui tendent à vérifier cette hypothèse.

Un problème de l'analyse de mutation est la détection des *mutants équivalents*. Un mutant équivalent est un mutant qui est sémantiquement équivalent au programme original. Les mutants équivalents faussent le score de mutation car ils ne peuvent être tués (par définition). La détection des mutants équivalents est un problème indécidable dans le cas général, même s'il est possible dans certains cas d'utiliser des heuristiques pour les détecter [Offutt 97].

Les opérateurs de mutation doivent injecter des erreurs qui correspondent à des modèles de fautes. Il est plus intéressant de qualifier la capacité d'une suite de tests à détecter des erreurs commises régulièrement par les programmeurs plutôt que des erreurs commises très rarement. Il faut donc identifier et analyser les erreurs commises régulièrement. Un modèle de faute décrit des types de fautes régulièrement rencontrées. Souvent ces modèles de fautes sont spécifiques à un langage particulier : ADA [Offutt 96], C [Barbosa 01], Java [Ma 02], etc.

De nombreux outils ont été proposés pour mettre en œuvre l'analyse de mutation : Mugamma [Kim 06], Mujava [Ma 05], CREAM system [Derezinska 08], NMutator [Baudry 05], JMutator [Baudry 06].

Pour cette thèse nous avons développé un outil de mutation pour les expressions de point de coupe AspectJ, AjMutator, basé sur des opérateurs proposés par Ferrari *et al.* [Ferrari 08]. Cet outil est présenté en détail dans le Chapitre 5.

1.3 Test de programmes orientés aspect

La programmation orientée aspect apporte de nouveaux défis pour le test de logiciels. Les nouveaux concepts introduits par la programmation orientée aspect introduisent aussi de nouveaux types d'erreurs qu'il faut détecter. Il est donc nécessaire d'adapter les différentes techniques de test.

1.3.1 Modèles de fautes pour la programmation orientée aspect

Différents auteurs ont proposé des modèles de fautes pour la programmation orientée aspect. Ces modèles de fautes décrivent les nouveaux types de fautes qui peuvent se produire dans des programmes orientés aspect, et permettent donc le développement de solution adaptées au test de programmes orientés aspect.

Alexander *et al.* [Alexander 04] ont proposé un modèle de faute pour la programmation orientée aspect, qui distingue différentes natures de fautes, selon leur source. Il y a quatre sources de faute dans un programme orienté aspect :

1. Une faute peut se trouver dans une partie du code de base qui n'est pas affectée par les aspects.
2. Une faute peut se trouver dans le code d'un aspect.
3. Une faute peut être une propriété émergente créée par une interaction entre le programme de base et un aspect.
4. Une faute peut être une propriété émergente créée par l'interaction de plusieurs aspects.

Les auteurs détaillent ensuite six types de fautes correspondant aux trois dernières natures de fautes. En effet la première nature n'est pas spécifique à la programmation orientée aspect. Les auteurs terminent en présentant six questions ouvertes pour le test de programmes orientés aspect.

Bækken et Alexander [Bækken 06] ont étendu et précisé ce modèle de faute en ce concentrant sur les fautes localisées dans les expressions de point de coupe. En effet ce type de faute est spécifique à la programmation orientée aspect et peut avoir de graves conséquences. Les langages de point de coupe peuvent être très différents selon les langages. Les auteurs se concentrent donc sur le langage de point de coupe d'AspectJ, mais le modèle de faute est utilisable pour des langages utilisant un langage de point de coupe proche de celui d'AspectJ.

McEachen et Alexander [McEachen 05] explorent les fautes possibles sur le long terme lors de l'évolution des programmes orientés aspect.

Ces différents travaux montrent que la programmation orientée aspect introduit de nouveaux types de fautes qui lui sont spécifiques. Il est donc nécessaire de développer des techniques de test spécifiques à la programmation orientée aspect, et ce aussi bien pour les greffons que pour les expressions de point de coupe. Dans le cadre de cette thèse, nous avons développé une technique de test spécifique pour le test des expressions de point de coupe. Cette approche est décrite dans le Chapitre 4.

1.3.2 Test du greffon

Les greffons sont tissés à différents endroits du programme, et donc une erreur dans un greffon peut avoir des conséquences dans plusieurs parties du programme. Les greffons peuvent aussi avoir de fortes interactions avec les contextes dans lesquels ils sont tissés. Enfin il n'est souvent pas possible de tester un greffon en dehors des contextes dans lesquels il est tissé.

Plusieurs travaux se sont intéressés à la génération de cas de test pour les greffons. Xie *et al.* [Xie 06b] proposent d'utiliser des outils de génération automatique de cas de test pour Java pour tester des greffons AspectJ. Les greffons sont tissés dans le code de base Java, les cas de test pour Java vont donc exécuter les greffons tissés. L'inconvénient d'une telle méthode est que le nombre de cas de test inutiles ou redondants peut être très important car les cas de test généré ont pour objectif de couvrir le code du programme de base et non le code des aspects. Ceci rend donc moins efficaces les suites de tests générées. Pour résoudre ce problème Xie et Zhao [Xie 06a] proposent un outil, nommé Aspectra, pour la détection des cas de test redondants ou inutiles.

Harman *et al.* [Harman 09] ont étendu Aspectra en utilisant des techniques évolutionnistes (*search-based*). Des données de test pour le programme de base sont automatiquement générées pour exécuter indirectement les greffons. L'objectif est de satisfaire une couverture structurelle des greffons (test boîte blanche). Cette approche a été implémentée dans un prototype, EvolutionaryAspectTester. Les auteurs ont aussi mené une étude empirique qui démontre l'efficacité des techniques évolutionnistes pour le test de programmes orientés aspect.

Une technique pour générer des cas de test pour les programmes orientés aspect est de générer les cas de test à partir de modèles du programme prenant en compte les modifications introduites par les aspects.

Zhao [Zhao 03] propose une approche basée sur les graphes de flots de données. L'objectif est de générer des cas de test unitaires pour des paires (*définition*,*utilisation*), c'est-à-dire qu'il faut générer des cas de test qui couvre un chemin de la définition d'une variable jusqu'à une utilisation de cette variable. L'approche proposée se base sur des graphes de flot de contrôle pour générer des cas de test qui prennent en compte les modifications introduites par les aspects.

Xu *et al.* [Xu 05, Xu 06] ont proposé une approche basée sur les diagrammes d'états. Un diagramme d'état représente l'état d'un objet selon les événements ayant lieu au cours de l'exécution. Les diagrammes d'état de chaque classe sont modifiés afin de prendre en compte les aspects tissés dans les classes. Puis des données de test sont générées afin de couvrir les chemins qui correspondent aux interactions entre le programme de base et les aspects.

Lemos *et al.*[Lemos 07] ont proposé un critère minimum de test pour le test des greffons. Ce critère, nommé *all-crosscutting-node* (tous les nœuds transverses), est satisfait si chaque greffon est exécuté à chaque point de jonction où il est tissé. Wedyan and Ghosh [Wedyan 08] appellent ce critère *couverture des points de jonction* et ont proposé un outil pour mesurer cette couverture. Si ce critère n'est pas validé par une suite de tests, il est possible que des problèmes d'interaction entre le greffon et un point de jonc-

tion particulier ne soient pas détectés.

Un défi pour le test unitaire des aspects est que, dans la plupart des langages, les greffons ne peuvent être exécutés seuls. Par exemple dans la syntaxe classique d'AspectJ, les greffons ne sont pas nommés et ne peuvent être référencé et ne peuvent donc être exécuté directement dans un cas de test unitaire. Pour les exécuter il est nécessaire d'exécuter le code où le greffon est tissé, et c'est ce que font les solutions généralement proposées.

Lopes et Ngo [Lopes 05] ont proposé une solution pour le test unitaire d'aspects. Ils proposent un outil de test unitaire des greffons, JamlUnit, basé sur le langage JAML. JAML est un langage orienté aspect basé sur Java. Les greffons sont des méthodes Java classiques et les expressions de point de coupe sont décrites avec XML. Ainsi les greffons peuvent être exécuté directement. JamlUnit permet de simuler le comportement du programme de base à l'aide de *mock objects*. Les greffons peuvent ainsi être testés unitairement, en isolation du programme de base.

Même s'il reste encore des défis pour le test du greffon, ces nombreux travaux apportent des solutions appropriées et originales. Dans le cadre de cette thèse nous nous somme concentré sur le test de l'expression de point de coupe plutôt que sur le test du greffon.

1.3.3 Test de l'expression de point de coupe

L'expression de point de coupe est un des points critiques dans les programmes orientés aspect. Une erreur dans une expression de point de coupe peut avoir des conséquences graves, par exemple en insérant un greffon à des points de jonction où il n'est pas attendu.

Anbalagan et Xie [Anbalagan 06, Anbalagan 08] ont proposé une solution pour aider le développeur à vérifier les expressions de point de coupe. L'idée de cette solution est de proposer des expressions de point de coupe proches de celles écrites par le développeur (en termes de points de jonction sélectionnés) afin qu'il vérifie si une de ces expressions de point de coupe correspond à son intention réelle. Afin de ne pas proposer trop d'expressions de point de coupe, les expressions de point de coupe candidates sont triées en mesurant leur *distance* par rapport à l'expression de point de coupe écrite par le développeur. La distance est calculée en comparant les ensembles de points de jonction sélectionnés par les expressions de point de coupe. Plus ses ensembles ont de point de jonction en commun, plus les expressions de point de coupe sont proches.

Ye et De Volder [Ye 08] ont proposé une solution proche. Cette solution s'appuie sur une extension pour l'IDE Eclipse, PointcutDoctor[1], qui signale au développeur les points de jonction qui n'ont pas été sélectionné par l'expression de point de coupe, mais qui sont proche de l'être (c'est-à-dire qu'il y a peu de changement à appliquer pour les sélectionner). PointcutDoctor indique aussi la raison pour laquelle un point de jonction n'a pas été sélectionné.

[1] `http://pointcutdoctor.sourceforge.net/`

Ces solutions offrent un support au développeur pour vérifier lors du développement s'il n'a pas commis une erreur dans l'expression de point de coupe. En revanche elles ne sont pas automatiques et lors d'une évolution de l'expression de point de coupe il est nécessaire de recommencer le processus sans pouvoir s'appuyer sur le travail de mise au point réaliser précédemment.

Dans le Chapitre 4 nous proposons une solution outillée au test d'expressions de point de coupe. Cette solution permet de définir des cas de test qui visent spécifiquement les erreurs dans les expressions de point de coupe. Les cas de test écrits avec cette approche n'échouent qu'en cas de faute dans une expression de point de coupe, ce qui facilite donc la localisation des fautes. Cette solution reposant sur des cas de test elle peut être automatisée.

1.3.4 Test de non-régression

Le test de non-régression a été appliqué à la programmation orientée aspect par différents auteurs. Xu et Rountev [Xu 07] ont adapté les travaux de Harrold *et al.* [Harrold 01] à AspectJ. Zhang *et al.* [Zhang 08] ont développé un outil d'analyse d'impact de changements pour AspectJ, Celadon. Il détecte les changements atomiques dans le code source du programme en comparant les arbres syntaxique abstraits des différentes versions. Puis les graphes d'appels des cas de test sont parcourus afin de détecter les cas de test affectés par les changements.

Ces analyses statiques sont proches de l'analyse que nous avons réalisé au cours de cette thèse, présentée Chapitre 3, mais elles différent car elles évaluent l'impact d'un changement arbitraire dans les aspects ou le programme de base alors que notre analyse évalue l'impact de l'introduction d'un aspect. Vidock, l'outil implémentant notre analyse, s'appuie sur l'expression de point de coupe pour localiser les méthodes impactées, tandis que les autres approches comparent deux modèles du programme (graphes de flot de contrôle, arbres syntaxique abstraits, etc.) afin de détecter quelles parties ont changé. Ceci entraîne de complexes comparaisons : d'après Rothermel et Harrold [Rothermel 97], leur algorithme est quadratique dans le pire des cas tandis que notre algorithme est linéaire avec la taille du graphe d'appel (nœuds et arêtes). Cet algorithme requiert aussi de connaître la couverture du modèle de la version précédente du programme par les cas de test. Vidock est plus spécifique et, dans le cas du tissage d'aspects, propose une solution efficace.

1.3.5 Analyse de mutation

Ferrari *et al.* [Ferrari 08] ont présenté des opérateurs de mutation pour les programmes orientés aspect. Ces opérateurs sont basés sur différents modèles de fautes [Alexander 04, Bækken 06], et concernent les greffons, les expressions de point de coupe, et les déclarations d'AspectJ. Ces opérateurs sont définis pour AspectJ, mais les auteurs comparent différents langages orientés aspect, et listent les opérateurs compatibles avec ces langages.

Au cours de cette thèse nous avons développé un outil, AjMutator, présenté dans le Chapitre 5, pour l'analyse de mutation des expressions de point de coupe AspectJ. Cet outil implémente les opérateurs de mutation pour les expressions de point de coupe présentés par Ferrari *et al.*.

Anbalagan et Xie [Anbalagan 08] ont développé un outil produisant des mutants d'expressions de point de coupe AspectJ. Même si cet outil utilise deux opérateurs de mutation, il n'effectue pas d'analyse de mutation. L'objectif n'est pas de qualifier une suite de test mais de proposer des expressions de point de coupe *candidates* au développeur afin qu'il puisse vérifier qu'une de ces *candidates* ne correspond pas à sont intention initiale.

1.4 Conclusion

La programmation orientée aspect est un nouveau paradigme ayant pour objectif de séparer les préoccupations transverses des préoccupations principales. Les préoccupations transverses sont encapsulées dans des aspects, qui sont tissés dans le programme de base, statiquement ou dynamiquement.

Ce nouveau paradigme introduit de nouveaux défis pour le test de logiciel. De nombreux types de fautes propres à la programmation orientée aspect ont en effet été proposés par Alexander *et al.* [Alexander 04], Bækken et Alexander [Bækken 06], ou McEachen et Alexander [McEachen 05]. Ces différents travaux montrent la nécessité de fournir des solutions nouvelles pour tester les programmes orientés aspect.

Afin de proposer des solutions adaptées à l'utilisation des aspects par les développeurs et capable de résoudre les problèmes concrets qu'ils peuvent rencontrer lors du développement de programmes orientés aspect, nous avons décidé de nous baser sur une étude empirique de programmes orientés aspect.

Dans le Chapitre 2, nous présentons une étude empirique que nous avons menée sur 38 projets libres écrits en AspectJ. Cette étude a pour but de comprendre comment les aspects sont utilisés sur des cas concrets, et de vérifier les différentes critiques qui ont été émises vis à vis de la programmation orientée aspect. Nous avons aussi mesuré différentes métriques sur des versions orientées objet et des versions orientées aspect d'un même système, afin de comparer les programmes orientés aspect et les programmes orientés objet.

Le développement de programmes orientés aspect est un processus incrémental où des aspects sont composés dans le programme de base. Il est donc possible de tester le programme de plusieurs façons, par exemple en testant le programme dans son ensemble une fois que les aspects ont été développés, ou de tester de manière incrémentale.

Dans le Chapitre 3 nous nous plaçons dans le cas où le programme est testé de manière incrémentale, d'abord le programme de base seul puis avec les différents aspects. Nous proposons une solution pour réutiliser un maximum de cas de test écrits pour le programme de base, afin de minimiser l'effort de test, et d'aider les développeurs à adapter les cas de test aux aspects.

Un aspect est composé d'un greffon et d'une expression de point de coupe. Plusieurs travaux ont été proposés pour le test des greffons. Xie *et al.* [Xie 06b] ont proposé une solution pour utiliser des outils de génération de cas de test pour Java avec AspectJ. D'autres travaux ont utilisé des modèles du programme prenant en compte les aspects pour générer des cas de test, comme Zhao [Zhao 03] qui a proposé une approche basée sur les graphes de flot de données, ou Xu *et al.* [Xu 05, Xu 06] qui ont proposé une approche basée sur les diagrammes d'états. Lemos *et al.*[Lemos 07] ont proposé un critère minimum de test pour le test des greffons. Lopes et Ngo [Lopes 05] ont proposé une solution pour le test unitaire de programmes orientés aspect.

Peu de travaux se sont intéressés au test de l'expression de point de coupe. Il est pourtant critique de s'assurer de la validité des expressions de point de coupe. En effet, l'expression de point de coupe décrit les points de jonction où un greffon est tissé, une erreur peut donc introduire des fautes à travers tout le programme. Anbalagan et Xie [Anbalagan 06, Anbalagan 08], et Ye et De Volder [Ye 08] ont proposés des solutions pour aider le développement des expressions de point de coupe, mais il n'existe pas, à notre connaissance, de travaux permettant un test précis et automatique des expressions de point de coupe.

Dans le Chapitre 4 nous proposons une solution pour le test d'expressions de point de coupe afin de définir des cas de test testant précisément les expressions de point de coupe. Contrairement à des cas de test générés avec d'autres techniques, ces cas de test n'échoue que lorsqu'une expression de point de coupe n'est pas correcte.

Chapitre 2

Études empiriques sur l'utilisation des aspects

Malgré un bon accueil à ses débuts, la programmation orientée aspect n'a pas été largement adoptée par l'industrie. Plusieurs problèmes et critiques de la programmation orientée aspect peuvent expliquer cette lente adoption. Il est par exemple reproché aux aspects d'être très techniques et de n'être vraiment utiles que dans un cas limité et définis de cas. La *fragilité de l'expression de point de coupe* a été identifié par Kellens *et al.* [Kellens 06]. Ce problème est rencontré lors de l'évolution d'un programme orienté aspect. L'expression de point de coupe peut sélectionner des nouveaux points de jonction introduits lors de l'évolution sans que ce soit l'intention initiale du programmeur, ou au contraire l'expression de point de coupe peut ne pas sélectionner de nouveaux points de jonction introduits par l'évolution. Un autre problème est que la programmation orientée aspect peut casser l'encapsulation des objets, ce qui peut entraîner des erreurs ou nuire à la sécurité du programme.

Afin de résoudre ces différents problèmes, de proposer des solutions innovantes, ou de développer des outils pour aider les développeurs à mieux appréhender la programmation orientée aspect, il est nécessaire de comprendre comment les développeurs utilisent la programmation orientée aspect, et quels sont les impacts de la programmation orientée aspect sur les propriétés d'un programme par rapport à la programmation orientée objet.

Nous avons mené une étude empirique en deux parties. La première partie porte sur 38 projets libres écrits avec AspectJ. Différentes métriques ont été mesurées sur ces projets, afin de comprendre l'utilisation de la programmation orientée aspect. Nous observons trois principales tendances : (1) les greffons sont tissés à peu d'endroits, (2) peu de greffons cassent l'encapsulation, et ceux qui le font ont des expressions de point de coupe très précises, (3) les expressions de point de coupe sont définies avec seulement la moitié des expressions disponibles.

L'objectif de la seconde partie est d'évaluer les différences de modularité et de testabilité entre la programmation orientée objet et la programmation orientée aspect. Ainsi nous mesurons un ensemble de facteurs sur deux implémentations d'un même sys-

tème, l'une étant écrite en Java et l'autre en AspectJ, et discutons les différences pour l'effort de test d'un programme orienté aspect.

La Section 2.1 détaille le protocole expérimental que nous avons appliqué. La Section 2.2 présente les différentes questions de recherche auxquelless nous cherchons à répondre. Puis la Section 2.3 présente les résultats de l'étude. Enfin la Section 2.4 conclut le chapitre.

2.1 Protocole expérimental

Cette section présente les données expérimentales et le protocole utilisés pour étudier empiriquement l'utilisation d'AspectJ.

2.1.1 Données expérimentales

Projets orientés aspect

Les données expérimentales pour l'étude de programmes orientés aspect consistent en 38 projets orientés aspect distribués sous licence libre avec leur code source. Nous avons récupéré ces projets sur des dépôts publics en juillet 2008. Les critères de sélection étaient les suivants :

1. les projets doivent être implémentés en AspectJ
2. le code source doit être disponible et public
3. les projets doivent pouvoir être compilés avec la version 1.5 du compilateur AspectJ
4. la taille du projet doit être d'au moins 10 classes et un aspect
5. les greffons dans le projet sont tissés à au moins un point de jonction

Les projets ont été trouvés sur Sourceforge[1], le dépôt de projets libres le plus populaire à ce moment-là. Sur 74 projets AspectJ, seulement 28 satisfaisaient à toutes les contraintes. Les autres projets ont été trouvés en utilisant Google Code Search[2]. Sur 29 projets trouvés, seuls 10 satisfaisaient toutes les contraintes.

La taille des projets varie de 1 116 à 80 818 lignes de code (LdC). On peut diviser les projets en trois parties selon leur taille : 36 % des projets sont de petite taille (entre 1 000 et 5 000 LdC), 36 % sont de moyenne taille (entre 5 000 et 20 000 LdC), et 28 % sont de grande taille (plus de 20 000 LdC). Les projets ont un total de 7 343 classes, 479 aspects, avec 522 greffons. Soixante-deux pour cent des projets ont entre un et 10 greffons, 25 % ont entre 10 et 30 greffons, et 13 % ont plus de 30 greffons.

[1] `http://sourceforge.net`
[2] `http://www.google.com/codesearch`

TAB. 2.1 – Nombre de classes, d'aspects, et taille des différentes versions du système HealthWatcher.

	Classes	**Aspects**	**Lignes de Code**
Java - V1	88	-	5 990
Java - V2	92	-	6 371
Java - V3	104	-	6 896
AspectJ - V1	91	11	5 690
AspectJ - V2	99	14	6 163
AspectJ - V3	111	18	6 414

Comparaisons de systèmes orientés objet et orientés aspect

Pour comparer la programmation orientée objets et la programmation orientée aspect, il est nécessaire d'effectuer les mesures sur des implémentations fonctionnellement équivalentes d'un même système.

Nous avons utilisé le système HealthWatcher[3]. Il s'agit d'une application web pour gérer des plaintes dans un hôpital. Ce système a été implémenté dans différents langages, dont Java et AspectJ. Il y a plusieurs versions différentes du système HealthWatcher, qui varient selon divers choix architecturaux. Nous avons utilisé trois versions différentes du système, chacune implémentée en Java et AspectJ – soit un total de six systèmes différents. Dans la deuxième version du système, le patron de conception «commande» est implémenté, tandis que dans la troisième le patron «commande» et le patron «état» sont implémentés [Gamma 95]. La Table 2.1 montre la taille des différentes versions du système HealthWatcher, en termes de nombre de classes, de nombre d'aspects, et de lignes de code.

2.1.2 Outils d'analyse

Plusieurs outils ont été utilisés pour analyser chaque projet et collecter les données nécessaires pour répondre aux questions posées.

Metrics : *Metrics*[4] est une extension pour l'environnement de développement Eclipse qui calcule différentes métriques sur des programmes Java. *Metrics* a été utilisé pour analyser les sources java des différents projets.

ABIS : *ABIS* [Munoz 08] est un outil qui vérifie les interactions entre les aspects et le programme de base. *ABIS* a été étendu pour s'appuyer sur ses capacités d'analyse des

[3] `http://www.comp.lancs.ac.uk/~greenwop/tao/implementation.htm`
[4] `http://metrics.sourceforge.net/`

programmes orientés aspect et extraire des données sur les aspects et leurs impacts sur le programme de base.

AjMetrics *AjMetrics* est un outil qui mesure des métriques sur les expressions de point de coupe et les programmes orientés aspect. Les caractéristiques des langages orientés aspect ont été abstraites dans un formalisme. Les métriques sont ensuite définies sur ce formalisme, ce qui permet de réutiliser ces définitions pour d'autres langages. Cette contribution de la thèse est détaillé, avec AjMetrics, dans le Chapitre 5.

2.1.3 Métriques

Différentes métriques ont été mesurées, à l'aide des outils d'analyses décrits précédemment pour comprendre l'utilisation des aspects et évaluer la testabilité d'AOP. Dans cette section nous présentons ces différentes métriques.

Lignes de code (*LdC*) : Le nombre de lignes de codes dans un projet.

Nombre de méthodes (*NOM*) : Le nombre de méthodes total dans un projet.

Nombre d'attributs (*NOA*) : Le nombre d'attributs dans un projet.

Nombre de greffons (*NOAD*) : Le nombre de greffons définis dans un projet. Seuls les greffons étant effectivement tissés sont comptabilisés.

Nombre de greffons invasifs (*NOIAD*) : Le nombre de greffons qui utilisent au moins une structure invasive.

Nombre de greffons utilisant une structure invasive (*NARI : Lecture, Écriture, Remplacement, Remplacement Conditionnel, Multiple, Argument*) : Pour chacune de ces structures invasives – voir Chapitre 1, page 25 –, le nombre de greffons l'utilisant.

Nombre de points de jonction par greffon (*NAJP*) : Pour chaque greffon on calcule le nombre de points de jonction où ce greffon est tissé. La somme de tous les *NAJP* est notée *CAJP*.

Nombre de points de jonction par greffon invasif (*NIJP*) : Le nombre de points de jonction où un greffon utilisant une structure invasive est utilisée. La somme de tous les *NIJP* d'un projet est notée *CIJP*.

Nombre de points de jonction calculable statiquement (*NOJP*) : Le nombre de points de jonction statiques où un greffon peut être tissé dans un projet.

Ratio greffons/méthodes (*AMR*) : Pour calculer cette métrique on divise le nombre de greffons par le nombre de méthodes. *AMR = NOAD / NOM*.

Ratio greffons invasifs/méthodes (*IAMR*) : Le nombre de greffons utilisant une structure invasive divisé par le nombre de méthodes dans un projet. *IAMR = NOIAD / NOM*.

Ratio de points de jonction sélectionnés (*JMR*) : Le nombre total de points de jonction sélectionnés dans un projet divisé par le nombre de points de jonction. *JMR = CAJP / NOJP*.

Ratio de points de jonction sélectionnés pour un greffon invasif (*IJMR*) : Le nombre total de points de jonction sélectionnés par les greffons invasifs dans un projet divisé par le nombre de points de jonction. *IJMR = CIJP /NOJP*.

Nombre d'expressions de point de coupe utilisant une primitive (*NPCD : and, or, not, execution, arg-dots, star, args, call, target, within, set, initialization, get, withincode, if, this, cflow, cflowbelow, staticinitialization, handler, adviceexecution, preinitialization*) : Pour chaque primitive d'expression de point de coupe, on calcule le nombre d'expression de point de coupe qui l'utilisent.

Couplage entre les objets (*CBO*) : Pour chaque classe *C* on compte le nombre de classes utilisées par *C* et le nombre de classe utilisant *C*. Les aspects insérant des greffons dans *C* sont aussi comptés.

Réponse à la classe (*RFC, RFC'*) : *RFC* compte le nombre des méthodes d'une classe ainsi que les méthodes directement invoquées par dans cette classe. *RFC'* est la clôture transitive, c'est-à-dire que *RFC'* compte toutes les méthodes pouvant être exécutées depuis une classe.

Couplage par envoi de messages (*MPC*) : le nombre d'invocations effectuées depuis une classe.

Manque de cohésion des méthodes (*LCOM*) : une valeur allant de 0 à 1, calculant le manque de cohésion d'une classe. Une valeur de 0 indique une parfaite cohésion : toutes les méthodes de la classe utilisent tous les attributs de la classe. Une valeur de 1 représente un manque total de cohésion : chaque attribut n'est référencé que par une méthode.

2.2 Questions de recherche

La motivation de ces analyses est de mieux comprendre l'utilisation des aspects dans les projets libres, ainsi que d'identifier les différences entre la programmation orientée objet et la programmation orientée aspect. Pour cela nous avons répondu à plusieurs questions de recherche.

Première question (Q1) : *Dans quelle mesure les aspects et les aspects invasifs sont-ils utilisés dans les projets orientés aspect ? Est-ce que l'utilisation de ces aspects dépend de la taille des projets ?* Cette question interroge sur l'usage qui est fait des aspects, et permet de savoir si les aspects sont utilisés de manière réduite, comme prédit par Steimann [Steimann 04, Steimann 06], ou si ils sont utilisés de manière intensive. Cette question s'intéresse aussi à la relation entre la taille d'un projet et l'utilisation des aspects.

Deuxième question (Q2) : *Dans quel mesure les aspects et les aspects invasifs sont-ils transverses au programme de base ? Cela dépent-il de la taille du système ?* L'objectif des aspects est d'implémenter une préoccupation transverse. Cette question interroge sur le nombre de points de jonctions où chaque greffon est tissé, et permet de savoir si les aspects correspondent vraiment à des préoccupations transverses, distribuées dans tout le code. Cette question permet de savoir si les aspects sont plutôt transverses, ou plutôt spécifiques à une partie du programme. Enfin, on s'interroge sur la relation entre la taille d'un projet et le nombre de points de jonction où sont tissés les greffons.

TAB. 2.2 – Statistiques des métriques *AMR* et *IAMR* (notation ingénieur)

	AMR	*IAMR*
Moyenne	27×10^{-3}	11×10^{-3}
Médiane	5×10^{-3}	4×10^{-3}
Centile 2,5	2×10^{-3}	1×10^{-3}
Centile 97,5	36×10^{-3}	6×10^{-3}
Minimum	300×10^{-6}	300×10^{-6}
Maximum	128×10^{-3}	74×10^{-3}
Écart type	39×10^{-3}	18×10^{-3}

Troisième question (Q3) : *Est-ce que les expressions de point de coupe utilisent toute l'expressivité permise par AspectJ ? Est-ce que les aspects invasifs sont tissés avec des expressions de point de coupe précises ?* C'est une question importante car le langage de point de coupe d'AspectJ permet de décrire des ensembles de points de jonction très compliqués. Il est intéressant de savoir si cette complexité est exploitée et maîtrisée par les développeurs, ou si au contraire ils s'en méfient.

Quatrième question (Q4) : *Est-ce que la programmation orientée aspect améliore la modularité ?* Un des objectifs de la programmation orientée aspect est d'améliorer la modularité du programme en retirant les préoccupations transverses du code de base. Si la modularité est effectivement améliorée, on peut s'attendre à ce que le nombre de méthodes, le nombre d'attributs, et le nombre de lignes de code diminuent. On peut aussi s'attendre à ce que la cohésion soit améliorée.

Cinquième question (Q5) : *Est-ce que la programmation orientée aspect améliore la testabilité ?* Il est intéressant de considérer les effets de la programmation orientée aspect sur la testabilité. Une meilleure modularité améliore souvent la testabilité, mais les effets de bord sur le couplage peuvent aussi détériorer la testabilité.

2.3 Résultats de l'analyse

2.3.1 Utilisation des aspects (Q1)

Nous avons analysé l'utilisation des greffons sous trois angles afin de répondre à la question Q1 : le nombre de greffons, l'évolution de ce nombre avec la taille des projets, et la partition des structures invasives parmi les greffons invasifs.

La Table 2.2 présente différentes statistiques pour les métriques *AMR* et *IAMR*. Les valeurs *AMR* ont été mesurées sur 36 projets et indiquent que le nombre de greffons par projet est très faible. Le nombre de greffons par méthode est très faible (la médiane

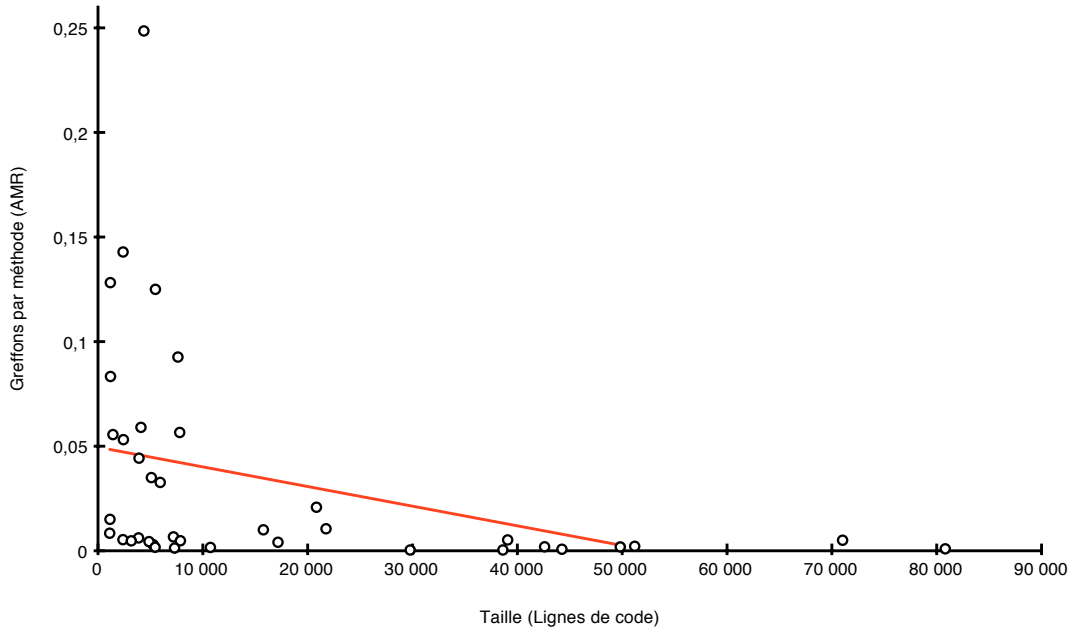

FIG. 2.1 – Nuage de points illustrant la relation entre *AMR* et la taille des projets

est de 5×10^{-3}). Soixante-huit pour cent des projets ont au maximum un greffon pour 37 méthodes, et 40 % des projets ont au maximum un greffon pour 200 méthodes.

Deux des 38 projets ont des valeurs exceptionnelles pour *AMR*. Ces deux projets sont de petite taille (moins de 5 500 LdC) et représentent des utilisations particulières de la programmation orientée aspect. Le premier projet utilise 27 greffons pour implémenter l'interface graphique et la gestion des exceptions. Le second projet utilise 84 greffons pour implémenter la persistance, la réplication, la gestion des exceptions et le *monitoring*.

Nous avons calculé *IAMR* pour les 21 projets contenant des aspects invasifs (soit 57 % des 38 projets). Un des projets ayant une valeur exceptionnelle pour *AMR* a aussi une valeur exceptionnelle pour *IAMR*, les valeurs *IAMR* ont donc été calculées sur 20 projets.

Les valeurs *IAMR* montrent que les greffons invasifs sont moins utilisés que les greffons normaux. Au maximum il y a un greffon pour 13 méthodes. Soixante-seize pour cent des projets avec des aspects invasifs ont au maximum un greffon pour 90 méthodes, et 47 % ont au maximum un greffon pour 250 méthodes.

La Figure 2.1 illustre la relation entre *AMR* et la taille des projets. *AMR* décroît avec la taille des projets. Un projet de grande taille n'implique donc pas un plus grand nombre de greffons. Les observations pour *IAMR* sont similaires à celles pour *AMR*.

Quatre projets ont des valeurs différentes de la tendance générale, deux petits et deux moyens. Ces projets ont une valeur *AMR* supérieure à 80×10^{-3}. Pour les deux projets de petite taille, cela s'explique par le faible nombre de méthodes (20 et 78). Les

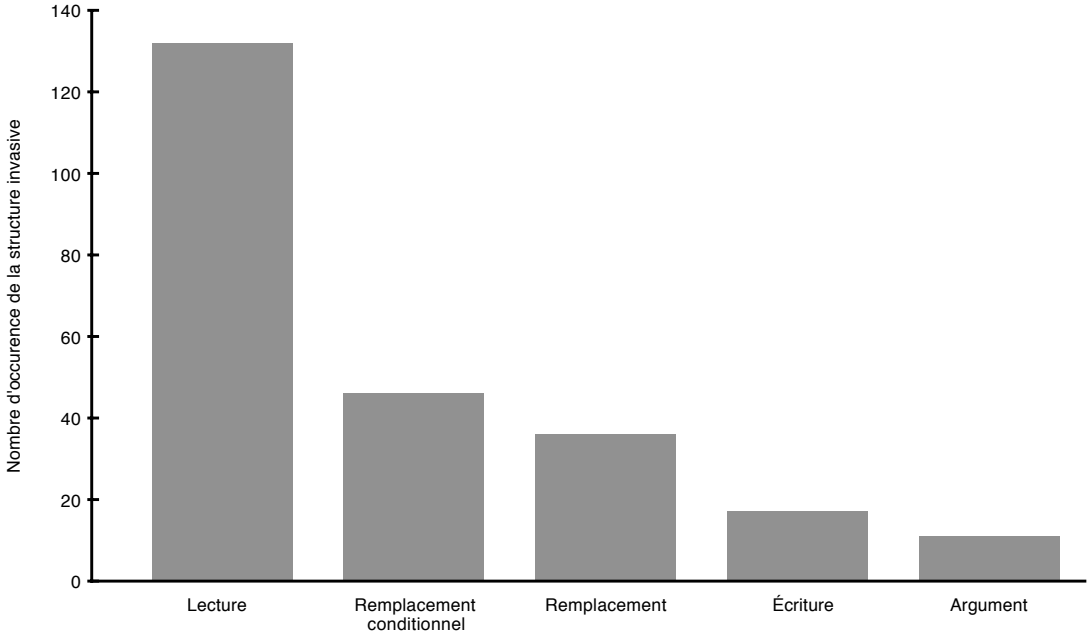

FIG. 2.2 – Histogramme des valeurs cumulées de la métrique *NARI*

projets de taille moyenne ont beaucoup de greffons. Dans l'un de ces projets, une préoccupation a été implémentée de plus de dix façons différentes avec 49 greffons différents. Il s'agit donc de cas particuliers.

Le seul projet ayant une valeur *IARM* supérieure à 70×10^{-3} est un petit projet ayant 14 greffons invasifs implémentant des fonctionnalités optionnelles.

Les Figures 2.2 et 2.3 montrent deux vues différentes de la métrique *NARI*. L'histogramme de la Figure 2.2 montre la somme des valeurs de *NARI* pour tous les projets, par structure invasive. La structure *multiple* n'est présente dans aucun projet et n'apparaît donc pas sur les graphiques.

L'histogramme montre que la structure invasive *lecture* est très utilisée, tandis que les autres structure le sont beaucoup moins. La boîte à moustaches de la Figure 2.3 montre aussi un net avantage à la structure *lecture*. Si en valeur absolue la structure *remplacement conditionnel* est plus utilisée (histogramme), la boîte à moustaches montre que ceci est dû à deux valeurs exceptionnelles, la structure *remplacement* étant plus utilisée dans les autres cas.

La structure `lecture` est très utilisée car elle n'affecte pas le flot de contrôle ou le flot données, les développeurs l'utilisent donc plus aisément. Dans une moindre mesure les structures *remplacement* et *remplacement conditionnel* sont aussi utilisées. Soixante-trois pour cent des greffons utilisant la structure *remplacement* sont dans trois projets (soit 14 % des projets) et 68 % des greffons utilisant la structure *remplacement conditionnel* sont dans trois projets. La structure *argument* est utilisée à 60 % dans un seul projet

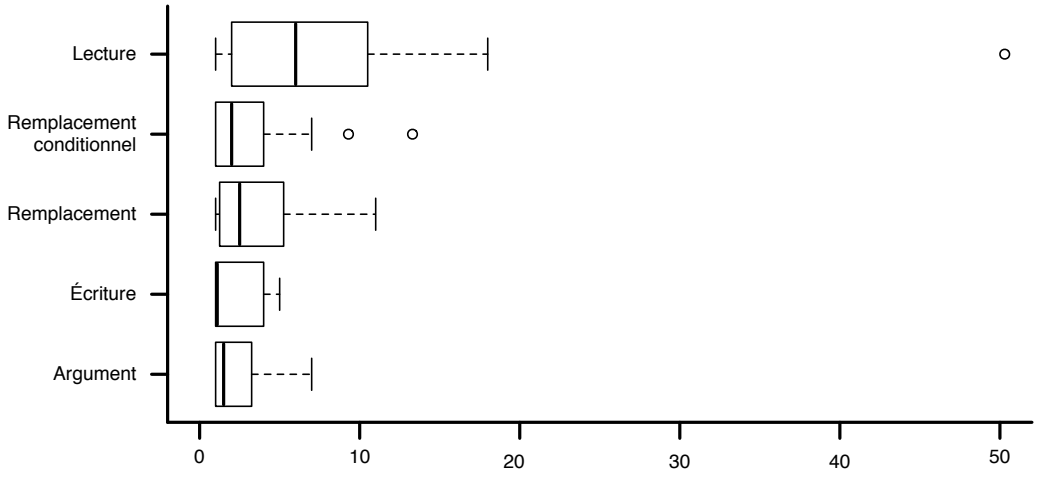

FIG. 2.3 – Boîte à moustache pour les métriques *NARI*.

pour traiter les arguments de requêtes sur un serveur web.

Ces résultats permettent plusieurs conclusions pour la question Q1 :

- *Les développeurs utilisent très peu de greffons pour implémenter les préoccupations transverses.* La valeur maximale pour *AMR* est très faible (128×10^{-3}), et la médiane n'est que de 5×10^{-3}.
- *Les développeurs utilisent peu de greffons invasifs.* Seulement 30 % des greffons utilisent une structure invasive. Ceci peut être expliqué par le fait que ces structures introduisent des effets de bord [Munoz 08] et donc les développeurs n'ont pas assez confiance dans ces structures pour les utiliser massivement. Cette hypothèse est confirmée par les observations de la métrique *NARI*, qui montrent que la structure *lecture*, qui n'a pas d'effet de bord, est de loin la plus utilisée.
- *Les projets de grande taille n'ont pas plus d'aspects.* Ceci contredit l'intuition que les projets plus grands ayant plus de méthodes devraient aussi avoir plus de greffons pour implémenter les préoccupations transverses.

2.3.2 Transversalité des aspects (Q2)

Dans cette section nous répondons à la deuxième question en analysant la proportion de points de jonction sélectionnés par les greffons et par les greffons invasifs.

La Table 2.3 montre différentes statistiques pour les métriques *JMR* et *IJMR*, respectivement le ratio de points de jonction sélectionnés pour les greffons, et le ratio de points de jonction sélectionnés pour les greffons invasifs. La Figure 2.4 compare *JMR* et *IJMR* à l'aide d'un histogramme.

La valeur maximale pour *JMR* est 92×10^{-3}, ce qui veut dire que 9,2 % des points de jonction du projet sont sélectionnés par les expressions de point de coupe. La moyenne est de 20×10^{-3} (2 % des points de jonction sélectionnés), et la médiane de 1×10^{-3} (0,1 % des points de jonction sélectionnés), ce qui montre que les greffons sont tissés à peu

TAB. 2.3 – Statistiques des métriques *JMR* et *IJMR* (notation ingénieur)

	JMR	*IJMR*
Moyenne	20×10^{-3}	3×10^{-3}
Médiane	1×10^{-3}	2×10^{-3}
Centile 2,5	$1,5 \times 10^{-3}$	700×10^{-6}
Centile 97,5	31×10^{-3}	3×10^{-3}
Minimum	100×10^{-6}	100×10^{-6}
Maximum	92×10^{-3}	13×10^{-3}
Écart type	24×10^{-3}	3×10^{-3}

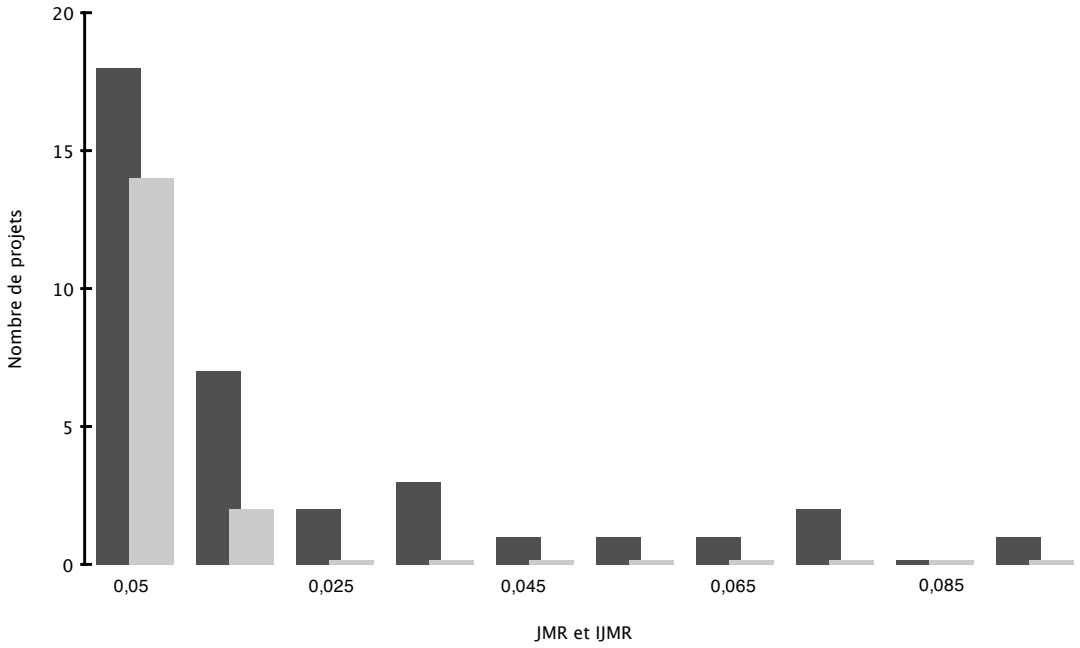

FIG. 2.4 – Histogramme présentant la répartition des projets en fonction de *JMR* et *IJMR*

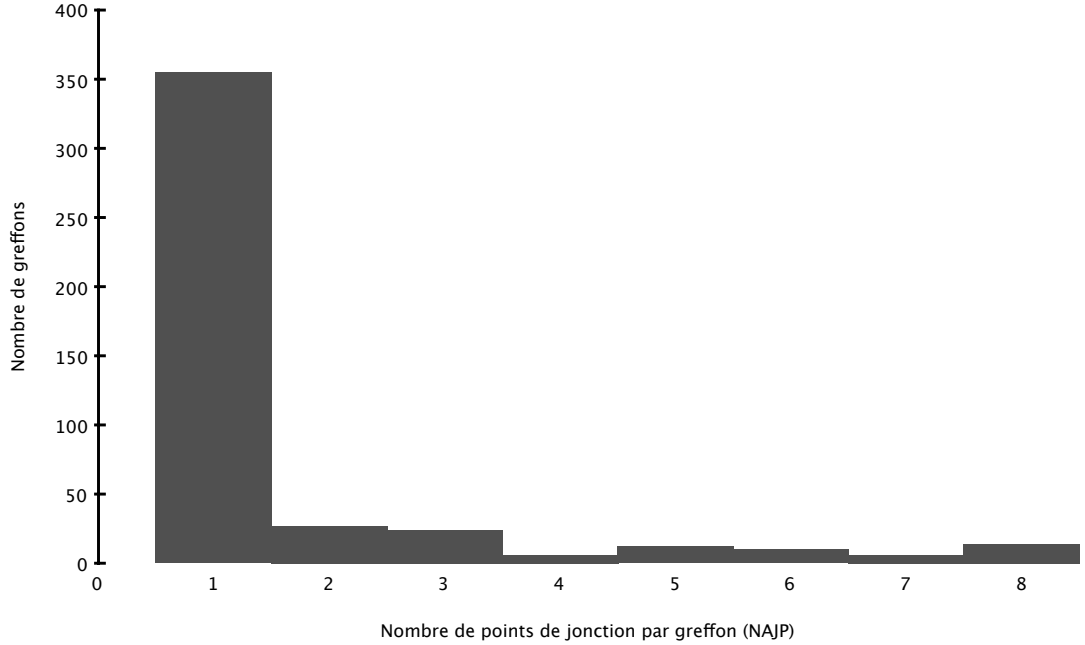

FIG. 2.5 – Histogramme présentant la répartition des greffons en fonction de *NAJP*

d'endroits. Cette tendance est clairement visible sur l'histogramme, où la majorité des projets (16, soit 41 % des projets) ont moins de 0,5 % des points de jonction sélectionnés et 27 % (10 projets) ont entre 0,5 et 2 % de points de jonction sélectionnés.

Deux projets ont des valeurs exceptionnelles pour *JMR* et ne sont pas pris en compte dans la Table 2.3. Le premier est un projet de grande taille (plus de 20 000 LdC) qui utilise les aspects pour mesurer la performance du système et des greffons sont tissés à chaque invocation de méthode. Le second est un projet de petite taille (moins de 5 500 LdC) qui utilise quatre greffons pour gérer les événements de sortie de l'interface graphique.

Les valeurs pour *IJMR* indiquent que les greffons invasifs sélectionnent moins de points de jonction que les autres greffons. Les valeurs pour *IJMR* sont toujours deux fois moins importantes que les valeurs pour *JMR*. La valeur maximale pour *IJMR* est 13×10^{-3}, soit 1,3 % des points de jonction sélectionnés. La moyenne et la médiane sont aussi très basses (respectivement 3×10^{-3} et 2×10^{-3}).

La Figure 2.5 présente la répartition des greffons selon la valeur de *NAJP*. Comme seulement 16 % des greffons sont tissés à plus de 10 points de jonction, l'histogramme ne considère que les valeurs de *NAJP* inférieures à 10. Cet histogramme montre que 69 % des greffons ne sont tissés qu'à un ou deux points de jonction. Ceci montre que la plupart des greffons étudiés sont très précis et qu'ils ne sont donc pas vraiment transverses.

La Figure 2.6 illustre la relation entre la taille des projets et *JMR*, à l'aide d'un nuage

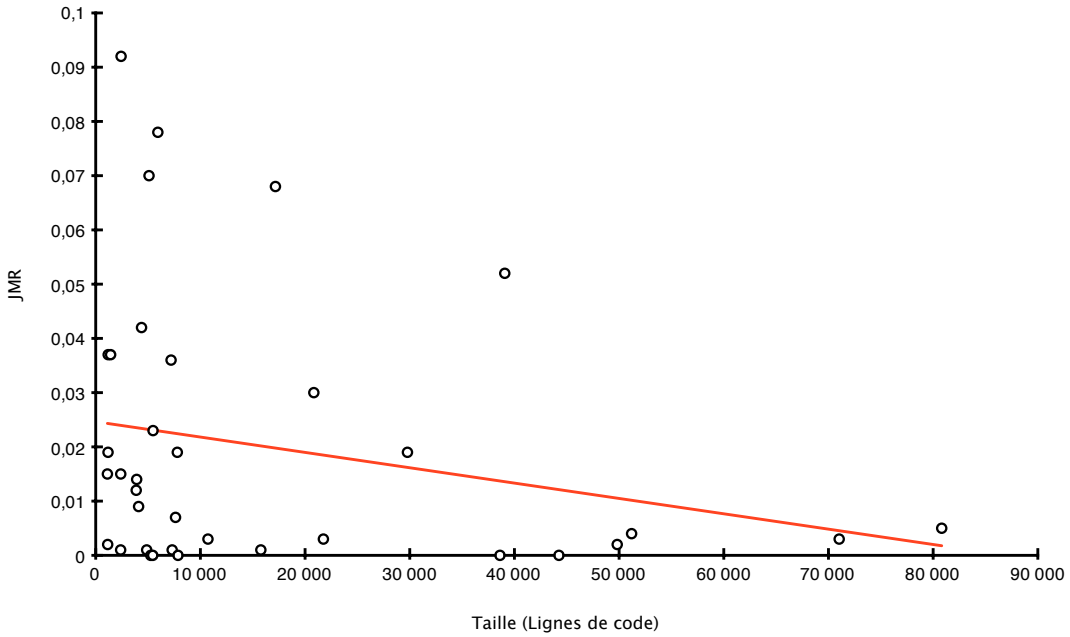

FIG. 2.6 – Nuage de points illustrant la relation entre *JMR* et la taille des projets

de points. La courbe montre que la tendance générale est une diminution de *JMR* avec l'augmentation de la taille des projets. Cependant, quelques projets diffèrent localement.

Ces résultats permettent plusieurs conclusions pour la question Q2 :

- *Les développeurs écrivent des greffons très précis qui sont tissés à peu de points de jonction.* La majorité des greffons (69 %) sont tissés à deux points de jonction ou moins. Ceci confirme le postulat de Steimann [Steimann 04, Steimann 06] : « les aspects sont peu nombreux et précis ».
- *Les développeurs utilisent les structures invasives avec une grande prudence* Les valeurs de *IJMR* montrent qu'en général les greffons invasifs sont tissés à très peu de points de jonction. Ceci peut être expliqué par les effets de bord introduits par les structures invasives.
- *Les aspects dans les projets de grande taille ne sont pas tissés à plus d'endroits différents.*

2.3.3 Utilisation de l'expression de point de coupe (Q3)

La Figure 2.7 présente les valeurs cumulées de NPCD pour tous les projets, et pour les projets contenant des aspects invasifs.

On observe qu'un ensemble de primitives d'expression de point de coupe sont très peu utilisées (moins d'un pour cent des expressions de point de coupe). Des primitives telles que *preinitialization*, *adviceexecution*, et *handler* sont utilisées au maximum dans quatre expressions de point de coupe sur 522 (moins de 0,8 %). Cinquante pour cent des

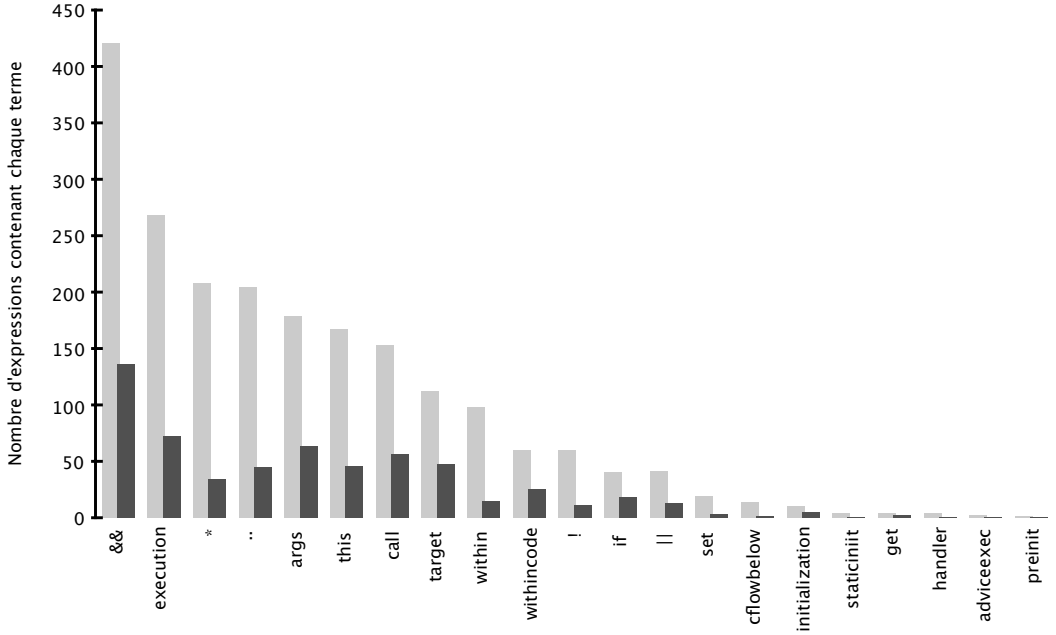

FIG. 2.7 – Histogramme des valeurs cumulées de NPCD pour tous les projets, et pour les projets contenant des aspects invasifs

primitives sont présentes dans moins de 8 % des expressions de point de coupe. Dans la plupart des cas, les développeurs utilisent donc moins de la moitié de l'expressivité du langage.

Les développeurs ont tendance à réduire l'ensemble des points de jonction sélectionnés plutôt qu'à l'augmenter. La conjonction («`&&`») est bien plus utilisée que la disjonction («`||`») ; 80 % des expressions de point de coupe utilisent la conjonction, contre huit pour cent pour la disjonction. La conjonction est l'intersection de différents ensembles de points de jonction, et restreint donc le nombre de points de jonction sélectionnés. La disjonction est l'union de différents ensembles de points de jonction, et augmente donc le nombre de points de jonction sélectionnés.

La primitive *execution* (51 % des expressions de point de coupe) est bien plus utilisée que la primitive *call* (29 %). La primitive *execution* sélectionne une (ou plusieurs avec l'utilisation de *jokers*) exécution d'une implémentation précise, tandis que la primitive *call* sélectionne tous les appels à une méthode. Ceci implique plusieurs différences. Un point de jonction sélectionné par *execution* se trouve dans la méthode exécutée, tandis qu'un point de jonction sélectionné par *call* se trouve dans la méthode appelante. En cas de polymorphisme, le greffon est toujours exécuté dans le cas de *call*, alors que dans le cas de *execution* il faut que la méthode ciblée soit effectivement exécutée. Ces différences rendent la primitive `execution` plus simple à utiliser ce qui explique sûrement sa plus grande utilisation.

On peut distinguer deux catégories de primitives dynamiques. Les primitives dynamiques *args*, *this*, et *target* sont présentes dans 20 à 35 % des expressions de point de coupe, tandis que les primitives dynamiques *if*, *cflow*, et *cflowbelow* ne sont présentes que dans quatre à huit pour cent des expressions de point de coupe. Les primitives *args*, *this*, et *target* permettent à la fois de restreindre dynamiquement les points de jonction sélectionnés, mais aussi de capturer des valeurs dans des paramètres du greffon. Or dans de nombreux cas ces primitives ne servent que pour les paramètres, et ne restreignent en pratique pas les points de jonction sélectionnés. Dans ce genre de cas, il est possible de réécrire l'expression de point de coupe sans utiliser de primitive dynamique. Les primitives *if*, *cflow* et *cflowbelow* sont bien plus difficiles à utiliser car elles imposent des restrictions complexes aux points de jonction, ce qui explique qu'elles soient très peu utilisées.

Les expressions de point de coupe associées à des greffons invasifs utilisent beaucoup moins de *jokers* que les autres expressions de point de coupe. Cela montre que les développeurs préfèrent énumérer les points de jonction dans le cas des greffons invasifs, afin de mieux contrôler où sont tissés ces greffons. On remarque aussi que les primitives dynamiques telles que *if*, *cflow* et *cflowbelow* ne sont presque jamais utilisées avec des greffons invasifs.

Ces résultats permettent plusieurs conclusions pour la question Q3 :

- *En général, les développeurs utilisent seulement la moitié de l'expressivité d'AspectJ.* La moitié des primitives d'expression de point de coupe est utilisée dans moins de huit pour cent des expressions de point de coupe.
- *Les développeurs écrivent des expressions de point de coupe très précises.* La disjonction, qui restreint le nombre de points de jonction, est largement utilisée tandis que la conjonction, qui augmente le nombre de points de jonction est très peu utilisée.
- *Les développeurs utilisent très peu les primitives dynamiques* if, cflow, *et* cflowbelow
- *Les expressions de point de coupe associées à des greffons dynamiques sont très précis.*

2.3.4 Modularité (Q4)

La Table 2.4 montre différentes statistiques sur l'évolution des métriques entre les versions Java et AspectJ du système HealthWatcher. Pour chaque classe où au moins un aspect est tissé, on fait la différence entre les métriques de la version Java et les métriques de la version AspectJ (un nombre positif indique que la métrique a augmenté avec AspectJ).

Le nombre de méthodes décroît dans les versions AspectJ. Pour la version 1 de HealthWatcher, la différence est faible (six méthodes de moins), tandis que pour les versions 2 et 3, la différence est plus importante (respectivement 25 et 28 méthodes de moins), avec une moyenne de près d'une méthode de moins par classe. Les versions AspectJ n'augmentent jamais le nombre de méthodes.

Le nombre d'attributs est presque constant. Dans les deux premières versions, une classe a un attribut de moins et une autre a deux attributs de plus. Pour la troisième version, une classe a un attribut de moins, une autre a un attribut de plus, et une dernière

TAB. 2.4 – Évolution du nombre de méthodes, du nombre d'attributs, du nombre de lignes de code, et de LCOM entre Java et AspectJ pour les différentes versions du système HealthWatcher.

Première Version				
	Méthodes	**Attributs**	**Lignes de code**	**LCOM**
Minimum	-4	-1	-101	-0,5
1er quartile	0	0	-12	0
Médiane	0	0	-9,5	0
3ème quartile	0	0	-1,75	0
Maximum	0	2	0	0,025
Moyenne	-0,25	0,042	-14,792	-0,021
Total	-6	1	-355	-0,492
Deuxième Version				
	Méthodes	**Attributs**	**Lignes de code**	**LCOM**
Minimum	-4	-1	-101	-0,5
1er quartile	0	0	-16	0
Médiane	0	0	-12	0
3ème quartile	-1	0	-5	0
Maximum	0	2	-2	0,025
Moyenne	-1,087	0,043	-17,478	-0,021
Total	-25	1	-402	-0,492
Troisième Version				
	Méthodes	**Attributs**	**Lignes de code**	**LCOM**
Minimum	-4	-1	-101	-1
1er quartile	-1	0	-16	0
Médiane	-1	0	-12	0
3ème quartile	0	0	-5	0,039
Maximum	0	2	0	0,182
Moyenne	-0,903	0,042	-15,290	-0,085
Total	-28	2	-474	-2,644

a deux attributs de plus. Le reste des classes est inchangé.

Le nombre de lignes de code décroît clairement. Dans les trois versions, presque 80 % des classes ont moins de lignes de code et aucune classe a plus de ligne de code. De plus les différentes versions AspectJ ont globalement moins de lignes de code malgré l'ajout des aspects ; la première version a 5 108 lignes de code en Java et 4 744 en AspectJ (respectivement 5 434 et 5 016, 5 734 et 5 360).

La cohésion est améliorée dans les différentes versions AspectJ. La métrique *LCOM*, qui mesure le manque de cohésion diminue de manière générale dans les trois versions.

Ces observations montrent qu'AspectJ permet effectivement d'améliorer la modularité. Un programme plus petit (en termes de nombre de méthodes et de lignes de code) ainsi qu'une meilleure cohésion sont en effet les signes d'une meilleure modularité.

2.3.5 Testabilité (Q5)

La Table 2.5 montre différentes statistiques sur l'évolution des métriques CBO, RFC, RFC', et MPC entre les versions Java et AspectJ du système HealthWatcher. Pour chaque classe où au moins un aspect est tissé, on fait la différence entre les métriques de la version Java et les métriques de la version AspectJ (un nombre positif indique que la métrique a augmenté avec AspectJ).

Dans les versions AspectJ, CBO et RFC' augmentent. Dans les deux premières versions, CBO augmente pour presque toutes les classes, tandis que le résultat est plus contrasté dans la troisième version avec tout de même une tendance à l'augmentation de CBO. CBO mesure le couplage entre les objets, donc dans les versions AspectJ les classes sont plus couplées. RFC' ne diminue que dans quelques classes, et dans les autres cas RFC' augmente énormément. RFC' mesure toutes les méthodes qui peuvent être exécutées en accédant à une classe donnée.

RFC et MPC décroissent globalement. RFC mesure le nombre de méthode qu'une classe invoque directement, tandis que MPC compte le nombre d'invocations.

Ces résultats peuvent sembler contradictoires. Une diminution de RFC et MPC indique une meilleure testabilité, tandis qu'une augmentation de CBO et RFC' indique une moins bonne testabilité. Cette contradiction peut être expliquée par un exemple simple.

Le Listing 2.1 montre une implémentation d'une bibliothèque en Java. Une politique d'accès a été implémentée, et avant d'ajouter ou de retirer un libre, les méthodes vérifient si l'accès est autorisé. La préoccupation de politique d'accès est une préoccupation transverse, et le Listing 2.2 montre une implémentation de la bibliothèque où un aspect encapsule le contrôle d'accès.

La Table 2.6 montre les métrique RFC, RFC', et MPC pour les deux implémentations. CBO ne peut être calculée sans connaître le système complet. Pour RFC', seule son évolution peut-être calculée.

RFC et MPC décroissent avec l'implementation AspectJ car dans chaque méthode, trois invocations ont été remplacées par l'insertion d'un greffon. L'amélioration de la modularité fait qu'il y a moins de méthodes invoquées directement. Mais les méthodes

TAB. 2.5 – Évolution de CBO, RFC, RFC', et MPC entre Java et AspectJ pour les différentes versions du système HealthWatcher.

Première Version				
	CBO	**RFC**	**RFC'**	**MPC**
Minimum	0	-7	-1	-32
1er quartile	1	-2	16	-9
Médiane	1	-1	26	-7,5
3ème quartile	3	0	27,250	1
Maximum	6	2	31	34
Moyenne	1,75	-1,333	20,958	-3,625
Total	42	-32	503	-87
Deuxième Version				
	CBO	**RFC**	**RFC'**	**MPC**
Minimum	0	-7	-1	-32
1er quartile	1	0	23	-11
Médiane	1	1	26	-8
3ème quartile	1	1	27	0,5
Maximum	7	1	48	34
Moyenne	1,217	-0,174	22,783	-5
Total	28	-4	524	-115
Troisième Version				
	CBO	**RFC**	**RFC'**	**MPC**
Minimum	-1	-9	-10	-32
1er quartile	0,5	-1	1,5	-9
Médiane	1	0	26	-2
3ème quartile	1	1	26,250	1
Maximum	7	1	48	34
Moyenne	0,710	-0,774	16,387	-3,677
Total	22	-24	508	-114

TAB. 2.6 – Métriques des deux implémentations de la bibliothèque.

	RFC	**RFC'**	**MPC**
Java	5	x	8
AspectJ	3	$x+1$	3

```
1  public class Library {
2     Collection<Book> books;
3
4     public void addBook(Book b) {
5        if (System.inMaintenance() || !AccessPolicy.isAuthorized
             ())
6           throw ExceptionFactory.
                createUnauthorizedAccessException(this,b);
7        books.add(b);
8     }
9
10    public void removeBook(Book b) {
11       if (System.inMaintenance() || !AccessPolicy.isAuthorized
             ())
12          throw ExceptionFactory.
                createUnauthorizedAccessException(this,b);
13       books.remove(b)
14    }
15 }
```

Listing 2.1 – Exemple d'une bibliothèque implémentée sans aspect

invoquées dans la version Java sont bien exécutées dans la version AspectJ. C'est pourquoi RFC' ne décroît pas, et au contraire augmente car un niveau de redirection a été ajouté. Ceci explique pourquoi, malgré une diminution de RFC et MPC, il y a plus de couplage, et donc une moins bonne testabilité.

Cet exemple illustre aussi comment la programmation orientée aspect implémente l'anti-patron *action à distance*. Un anti-patron est une mauvaise pratique de programmation, au contraire des patrons de conception qui sont des bonnes pratiques. Dans l'anti-patron *action à distance*, le comportement d'une classe varie selon des opérations dans une autre classe, qui sont difficilesvoire impossibles à identifier. Simplement en regardant le code source de la classe `Library` du Listing 2.2, il n'est pas possible de savoir que le comportement des méthodes `addBook` et `removeBook` peut ne pas s'exécuter. Cet anti-patron réduit la testabilité.

2.3.6 Menaces à la validité

Dans cette section nous discutons des menaces à la validité interne et à la validité externe.

Les menaces à la validité interne sont les menaces qui peuvent biaiser les observations effectuées durant cette étude. Le nombre de greffons, de points de jonction, et de points de jonction sélectionnés sont détecté à l'aide du compilateur AspectJ actuel. Il est possible que des différences entre versions du compilateur, ou des bugs, induisent une

```
1  public class Library {
2     Collection<Book> books;
3
4     public void addBook(Book b) {
5        books.add(b);
6     }
7
8     public void removeBook(Book b) {
9        books.remove(b);
10    }
11 }
12
13 public aspect AccessPolicy {
14    before(): execution(void Library.*(Book)) {
15       Library l = getJoinpoint().getTarget();
16       if (System.inMaintenance() || !AccessPolicy.isAuthorized
             ())
17          throw ExceptionFactory.
                createUnauthorizedAccessException(l,b);
18    }
19 }
```

Listing 2.2 – Exemple d'une bibliothèque implémentée avec aspect

différences entre l'expérience du développeur au moment où il a écrit le programme et nos mesures.

Les menaces à la validité externe sont les menaces qui peuvent biaiser la généralisation des résultats obtenus. Nous avons étudié 38 projets libres écrits en AspectJ, nous ne savons pas à quel point nos observations peuvent être généralisées à d'autres langages orientés aspect, ou à des projets industriels. Dans le cas de la comparaison entre la programmation orientée objet et la programmation orientée aspect, nous n'avons étudié qu'un système, et il est possible que les résultats soient différents avec d'autres systèmes.

2.4 Conclusion

Nous avons étudié 38 projets AspectJ sous licence libre, de taille petite (moins de 5 000 lignes de code) à grande (plus de 20 000 lignes de code), pour un total de 479 aspects et 522 greffons. L'objectif était de mieux comprendre l'utilisation qui est faite de la programmation orientée aspect, des aspects invasifs ou encore du langage d'expression de point de coupe.

Nos observations révèlent que les développeurs utilisent peu de greffons et que ces greffons sont peu transverses (ils sont tissés à peu d'endroits). Les développeurs écrivent peu de greffons invasifs, et lorsqu'ils le font, ces greffons sont très spécifiques. Environ la moitié des expressions du langage d'expression de point de coupe d'AspectJ n'est pratiquement pas utilisée par les développeurs.

Il existe plusieurs façons d'expliquer ces observations. Les développeurs peuvent trouver qu'il est difficile de raisonner sur des modules transverses, et se limitent donc à des types d'aspects qu'ils maîtrisent. Il se peut que le langage AspectJ n'offre pas la flexibilité qui permettrait aux développeurs de mieux modulariser les programmes. Les développeurs peuvent aussi se méfier des structures invasives qui permettent de rendre modulaire les préoccupations transverses car ces structures peuvent avoir des effets de bord difficiles à maîtriser. Enfin, les aspects peuvent nuire à la testabilité ou à la maintenance (fragilité de l'expression de point de coupe), ce qui freine l'utilisation des aspects.

En comparant plusieurs implémentations, en Java – orienté objet – et en AspectJ – orienté aspect –, nous avons constaté que les versions AspectJ avait une meilleur modularité, mais une moins bonne testabilité. Ainsi la programmation orientée aspect remplit son contrat et modularise bien le programme en séparant les préoccupations transverses. Cependant, la programmation orientée aspect diminue la testabilité en augmentant le couplage. En effet, un aspect est couplé avec toutes les classes dans lesquelles il est inséré et doit donc être testé dans tous les contextes où il est tissé.

Différents problèmes ralentissent l'adoption massive de la programmation orientée aspect et il est nécessaire de fournir des solutions outillées à ces problèmes. En particulier la diminution de la testabilité rend nécessaire le développement d'outils d'analyse et de techniques de test pour les aspects. Colyer *et al.* [Colyer 06] ont indiqué que des

outils d'analyses étaient nécessaires pour l'adoption de la programmation orientée aspect.

Afin d'améliorer la testabilité nous avons développé deux solutions principales. Dans le Chapitre 3 nous nous plaçons dans le cadre d'une approche incrémentale du test de programmes orientés aspect où le programme de base est d'abord testé sans les aspects, puis les aspects sont ajoutés. Nous proposons une analyse statique permettant de réutiliser le maximum de cas de test possible, les autres devant être modifiés afin de prendre en compte les aspects. Il est ainsi plus facile de prendre en compte le couplage induit par les aspects.

Dans le Chapitre 4 nous présentons une approche pour le test automatique des expressions de point de coupe en AspectJ. Une erreur dans l'expression de point de coupe peut avoir de graves conséquences, c'est pourquoi les développeurs utilisent des expressions de point de coupe très précises afin de pouvoir contrôler plus facilement les effets du tissage. Nous proposons donc une solution permettant de définir des oracles précis ayant pour but de détecter des erreurs dans les expressions de point de coupe.

Chapitre 3

Analyse de l'impact des aspects sur les cas de test

Lors du test d'un programme orienté aspect, deux stratégies sont possibles. La première consiste à tester le programme dans sa globalité, c'est-à-dire avec les aspects. Une seconde stratégie consiste à tester le programme de manière incrémentale, en commençant par le programme de base.

Nous nous plaçons dans le cas où le programme de base est testé d'abord, puis les aspects sont ajoutés. L'avantage de cette solution est que le programme de base est plus testable que le programme entier. Le programme de base est plus petit que le programme entier : il y a moins de méthodes, pas d'aspects et donc moins de couplage. Aussi, le programme de base est un programme orienté objet normal et est a priori plus testable qu'un programme orienté aspect comme vu précédemment.

Il existe deux problèmes avec cette stratégie. D'abord, si les aspects introduisent trop de changements à trop d'endroits différents, la majorité des cas de test écrits pour le programme de base aura besoin d'être réécrite, ce qui entraîne un surcoût trop important. Ensuite, même si les changements ne sont pas trop nombreux il reste difficile d'identifier les cas de test qui doivent prendre en compte ces changements.

Concernant le nombre de changements introduits par les aspects, nos études ont montré, voir Chapitre 2, que le nombre de points de jonction sélectionnés est très faible pour une majorité des expressions de point de coupe. La stratégie proposée semble donc raisonnable, car on peut s'attendre à ce qu'une grande partie des cas de test pour le programme de base puisse être réutilisée pour le programme avec aspects.

Le second problème nécessite une solution adaptée. Par exemple, dans le cas de la vente aux enchères, on peut écrire le cas de test présenté sur le Listing 3.1. Ce cas de test crée deux utilisateurs (un vendeur et un acheteur). Le vendeur met en vente un objet à un prix minimum de 30 €, et l'acheteur enchérit 40 €. Ensuite le cas de test termine la vente et vérifie que le montant de la vente, dans ce cas 40 €, a bien été transféré du compte de l'acheteur vers le compte du vendeur. Si on ajoute l'aspect `AltBid` (voir Section 1.1.2, page 13) dont un extrait est présenté sur le Listing 1.1, page 15, le montant de la vente devient le montant de la deuxième meilleure enchère (ou l'enchère minimum

```
1  public class TestAuction {
2     @Test
3     public void testClose() {
4        User buyer = new User(...);
5        buyer.getAccount.add(40);
6        User seller = new Buyer(...);
7        Auction auction = new Auction(seller,"description",30);
8        auction.open();
9        auction.placeBid(buyer,40);
10       auction.close();
11       assertEquals(40,
12          seller.getAccount().getBalance());
13       assertEquals(0,
14          buyer.getAccount().getBalance());
15    }
16 }
```

Listing 3.1 – Exemple de cas de test pour la classe `Auction`

s'il n'y a qu'une enchère) plus 10 %. Donc dans le cas de test, le montant de la vente n'est plus 40 €, mais 33 €. Le cas de test qui passait avant, ne passe plus après l'introduction de l'aspect, puisque l'assertion de la ligne 11 n'est pas validée. Cet échec est dû au fait que l'aspect a modifié le flot de données de la méthode `Bid.getAmount`, en modifiant la valeur de retour. Pourtant le cas de test n'exécute pas directement `Bid.getAmount`. Cet impact des modifications du flot de contrôle ou du flot de données sur les cas de test est difficile à détecter, et vérifier individuellement chaque cas de test peut être très coûteux.

Afin d'identifier automatiquement les cas de test nécessitant d'être adaptés ou supprimés et les cas de test pouvant rester inchangés, nous proposons une analyse statique de l'impact des aspects sur les cas de test. Le but de cette analyse est d'identifier les cas de test qui peuvent exécuter un greffon tissé dans le programme de base. Une des particularités de la programmation orientée aspect est qu'il est possible de connaître statiquement où un greffon est tissé, et donc quelles sont les modifications apportées par un aspect.

Dans un premier temps nous identifions les méthodes où des greffons ont été tissés ; ces méthodes sont dites *impactées*. Ensuite, grâce à une analyse statique des cas de test, nous déterminons quels sont les cas de test qui peuvent exécuter ces méthodes impactées ; ces cas de test sont dit *impactés*.

Une analyse statique évite l'exécution des cas de test non-impactés. Les cas de test non-impactés ne sont affectés par aucun effet de bord introduit par des aspects. Le résultat de ces cas de test ne change donc pas avec l'introduction des aspects, et il n'est donc pas nécessaire des les exécuter. Si ces cas de test non impactés sont nombreux, ne pas les exécuter peut faire gagner beaucoup de temps. L'analyse statique permet

aussi d'effectuer l'analyse sur un programme qui n'est pas complet ou qui comporte des fautes empêchant l'exécution normale du programme.

Nous avons implémenté l'analyse d'impact dans un outil nommé Vidock, disponible sur Internet [1]. Vidock est une extension pour l'environnement de développement Eclipse. L'analyse s'intègre parfaitement à l'environnement en s'exécutant de manière transparente pour l'utilisateur.

Dans ce chapitre nous détaillons l'analyse statique proposée. La Section 3.1 détaille la façon dont les méthodes impactées puis les cas de test impactés sont sélectionnés. Puis la Section 3.2 décrit comment l'analyse a été implémenté. Enfin la Section 3.3 montre avec de cas d'étude, l'intérêt de l'analyse statique proposée.

3.1 Sélection des cas de test impactés

Nous proposons une analyse d'impact afin de déterminer quels sont les cas de test impactés par l'introduction d'aspects. Cette analyse permet de distinguer les cas de test ne nécessitant ni d'être modifiés ni d'être exécutés des cas de test nécessitant une attention particulière lors du re-test du programme après l'introduction d'un aspect.

Cette analyse est implémentée dans l'outil Vidock. Vidock analyse d'abord l'expression de point de coupe et sélectionne l'ensemble des méthodes impactées par un aspect. Ensuite, Vidock analyse les cas de test et sélectionne l'ensemble des cas de test impactés. Vidock s'appuie sur AJDT [2] (Aspectj Development Tools), un outil de développement pour AspectJ, et sur Spoon [Pawlak 06], un outil d'analyse statique pour Java.

La Figure 3.1 présente une vue d'ensemble du processus. D'abord, Vidock récupère des informations de AJDT et de Spoon. Puis, il sélectionne toutes les méthodes où au moins un aspect est tissé (a). Ces méthodes sont appelées *méthodes impactées*. Parallèlement, Vidock construit un graphe d'appels statique (SCG, *Static Call Graph*) pour chaque cas de test ((b) et (c)). Finalement, Vidock calcule l'ensemble des cas de test impactés, en recoupant l'ensemble des méthodes impactées et les graphes d'appels statiques (d).

3.1.1 Sélection des méthodes impactées

La première partie de l'analyse statique sélectionne les méthodes impactées par le tissage d'aspects. La sélection des méthodes est divisée en deux étapes, (1) nous récupérons l'ensemble des points de jonction sélectionnés par les expressions de point de coupe (réalisé par AJDT), et ensuite (2) nous calculons l'ensemble des méthodes où se trouvent ces points de jonction (réalisé par Vidock).

La sélection des méthodes impactées par les aspects pose deux problèmes. Il faut gérer les expressions de point de coupe dynamiques, et il faut déterminer la méthode correspondant à un point de jonction.

[1] `http://www.irisa.fr/triskell/Softwares/protos/vidock/`
[2] `http://www.eclipse.org/ajdt/`

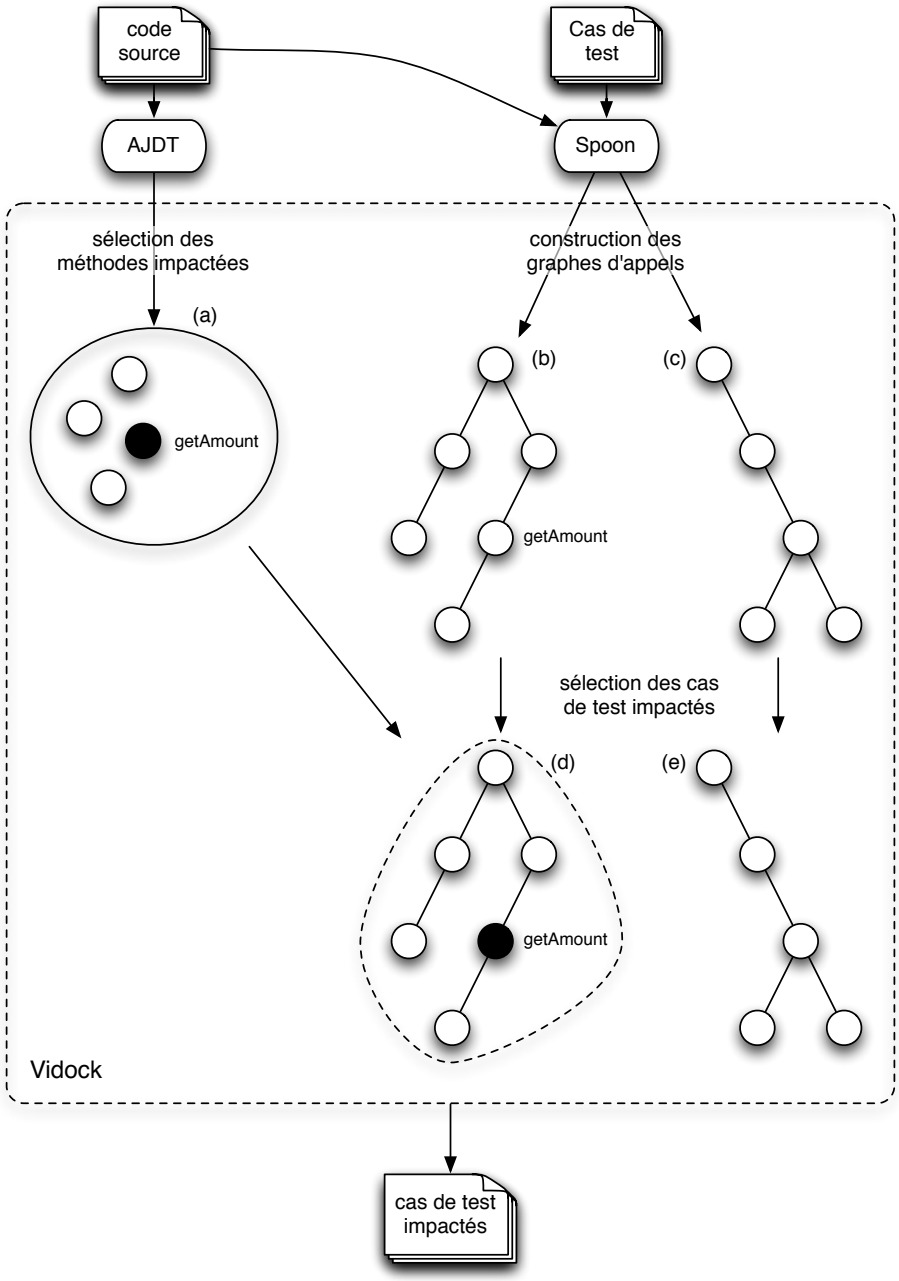

FIG. 3.1 – Vue d'ensemble du processus.

Expressions de point de coupe dynamiques

Les expressions de point de coupe dynamiques sont des expressions de point de coupe avec une partie dynamique. Toutes les expressions de point de coupe ont une partie statique qui sélectionne un certain nombre de points de jonction correspondant à des positions dans le code source. La partie dynamique d'une expression de point de coupe ajoute une contrainte qui doit être vérifiée dynamiquement pour que le greffon soit exécuté. L'ensemble effectif de points de jonctions est donc inclus dans l'ensemble de points de jonction sélectionnés par la partie statique.

Le Listing 1.1 montre un exemple d'expression de point de coupe dynamique, `getAmount`. Cette expression de point de coupe sélectionne les appels à la méthode `Bid.getAmount` effectués dans le flot de contrôle de la méthode `Open.close`. La condition d'exécution du greffon dépend du flot d'exécution, et n'est donc pas calculable statiquement. Il est même possible que le même appel à `Bid.getAmount` soit sélectionné dans un cas de test mais pas dans un autre. Il faut donc faire une sur-approximation.

AJDT fait une sur-approximation de l'ensemble des points de jonction en ne considérant que la partie statique de l'expression de point de coupe. Formellement, cette sur-approximation peut être décrite comme suit. Soit P un programme, C l'ensemble de toutes les expressions de point de coupe déclarées dans P, et J l'ensemble de tous les points de jonction. Soit $f_{pc} : C \rightarrow \mathcal{P}(J)$ la fonction qui retourne l'ensemble des points de jonction sélectionnés par une expression de point de coupe. Soit $f_{sur} : C \rightarrow \mathcal{P}(J)$ la fonction qui retourne la sur-approximation de l'ensemble des points de jonction.

$$\forall c \in C, f_{pc}(c) \subseteq f_{sur}(c)$$

Dans le cas où l'expression de point de coupe est statique, $f_{pc} = f_{sur}$.

La Figure 3.2 illustre cette sur-approximation. L'expression de point de coupe `getAmount` (A) sélectionne l'ensemble de points de jonctions $f_{pc}(A) = J_d$. Comme vu précédemment, il est impossible de calculer J_d statiquement. L'expression de point de coupe statique B sélectionne un ensemble de points de jonctions $f_{pc}(B) = J_s$ plus grand, tel que $J_d \subset J_s$. En pratique, l'ensemble calculé par AJDT est $f_{sur}(A) = J_s$.

Association d'une méthode à un point de jonction

Le second problème pour l'analyse statique de l'impact d'un aspect est la sélection effective des méthodes impactées. Une méthode impactée est une méthode liée à un point de jonction sélectionné par une expression de point de coupe. La spécification de la sélection des méthodes impactées est décrite ci-dessous.

Soit M l'ensemble des méthodes dans un programme P. Soit $g : J \rightarrow M$ la fonction qui à tout point de jonction associe la méthode liée.

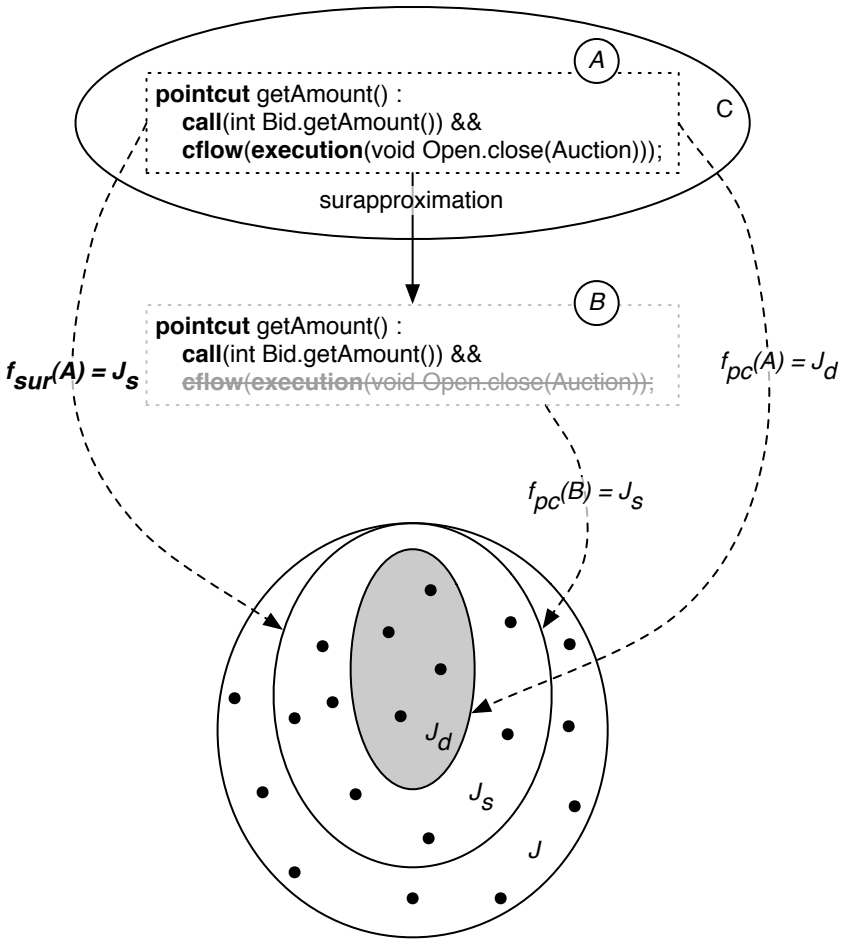

FIG. 3.2 – Sur-approximation de l'ensemble des points de jonction sélectionnés.

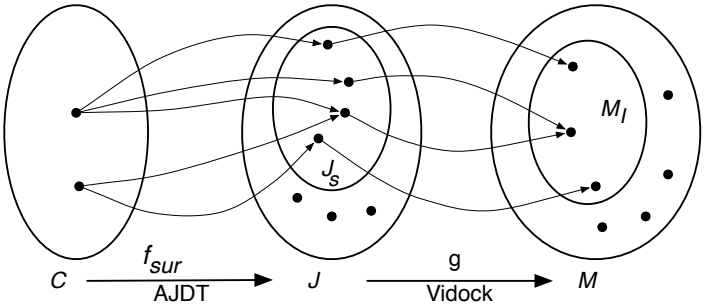

FIG. 3.3 – Vue schématique de la définition d'une *méthode impactée*.

$$g(j) = \begin{cases} \text{la méthode appelante} & \text{si } j \text{ est un point de jonction de type } \texttt{call} \\ \text{la méthode exécutée} & \text{si } j \text{ est un point de jonction de type } \texttt{execution} \\ \text{le constructeur} & \text{si } j \text{ est un point de jonction de type } \texttt{initialization} \\ \text{la méthode accédant au champ} & \text{si } j \text{ est un point de jonction de type } \texttt{set} \text{ ou } \texttt{get} \end{cases}$$

Puis, d'après g, l'ensemble $M_I \subseteq M$ des méthodes impactées est défini comme :

$$M_I = g(f_{sur}(C))$$

La Figure 3.3 décrit comment l'ensemble M_I des méthodes impactées est obtenu à partir de la combinaison de f_{sur} et g. D'abord f_{sur} fournit l'ensemble des points de jonction J_s à partir de l'expression de point de coupe A (`getAmount`), $J_s = f_{sur}(A)$. Cette partie est effectuée par AJDT. Puis, l'ensemble $M_I = g(J_s)$ est calculé par Vidock, qui implémente g. Dans ce cas $g(j_1) =$ `Open.open`, $g(j_2) = g(j_3) =$ `Bid.finalize` et $g(j_4) =$ `Bid.bid`. Ces méthodes sont impactées.

3.1.2 Cas de test impactés

La deuxième étape de l'analyse d'impact consiste à détecter les cas de test impactés par l'introduction des aspects. Nous cherchons à savoir si un cas de test peut atteindre une méthode impactée, afin de savoir si son flot de contrôle ou son flot de données peuvent être modifiés. D'abord, nous devons connaître les méthodes atteignables pour chaque cas de test. C'est pourquoi nous construisons un graphe d'appels statique pour chaque cas de test.

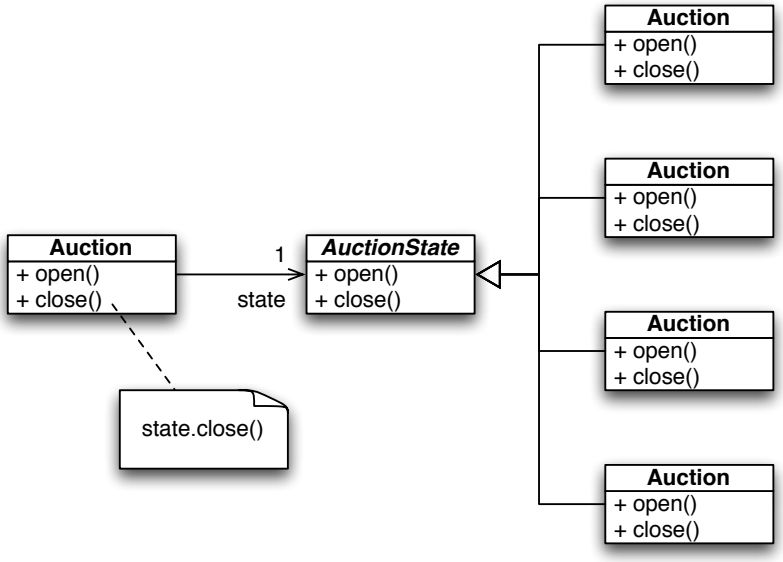

FIG. 3.4 – Diagramme de classes illustrant l'implémentation du patron de conception «état».

Graphe d'appels statique Un graphe d'appels statique est un triplet (V, E_a, E_p) où :

- V est l'ensemble des nœuds du graphe, chacun représentant une méthode du programme.
- $E_a \subseteq V \times V$ est l'ensemble des arêtes représentant les appels explicites de méthode. Si $(m_1, m_2) \in E_a$, alors la méthode représentée par le nœud m_1 appelle explicitement la méthode représentée par le nœud m_2.
- $E_p \subseteq V \times V$ est l'ensemble des arêtes représentant les appels potentiels de méthode. Si $(m_1, m_2) \in E_a$, alors la méthode représentée par le nœud m_1 peut potentiellement appeler la méthode représentée par le nœud m_2.
- $E_a \cap E_p = \emptyset$

Appels explicites et potentiels Une méthode m_1 appelle explicitement m_2 si m_2 est invoquée explicitement dans le code de m_1. La méthode m_1 appelle potentiellement m_2 si elle appelle explicitement une méthode redéfinie par m_2. C'est un appel *potentiel* car, à cause du polymorphisme, il n'est pas possible de savoir statiquement quelle méthode est effectivement exécutée. Par exemple, la classe `Auction` utilise le patron de conception «état» (illustré par la Figure 3.4), quand `Auction.close` est appelée, la méthode `close` de l'objet `State` courant est appelée, mais la méthode qui est effectivement exécutée peut être n'importe laquelle des méthodes redéfinissant `close` dans `Pending`, `Open`, `Closed` et `Cancelled` (les quatre états possibles pour une enchère).

Le listing 3.2 montre un exemple de cas de test JUnit du système de vente aux enchères. Le cas de test teste la méthode `close` de la classe `Auction`. D'abord une vente est créée et deux utilisateurs placent une enchère. Puis la vente est fermée et l'oracle

```
1  public class TestAuction {
2        @org.junit.Test
3        public void testClose() {
4              Auction a = new Auction(..);
5              a.open();
6              a.bid(user1,30);
7              a.bid(user2,50);
8              a.close();
9              assertTrue(a.isClosed());
10             assertEquals(user2,a.getBuyer());
11     }
12 }
```

Listing 3.2 – Exemple de cas de test JUnit.

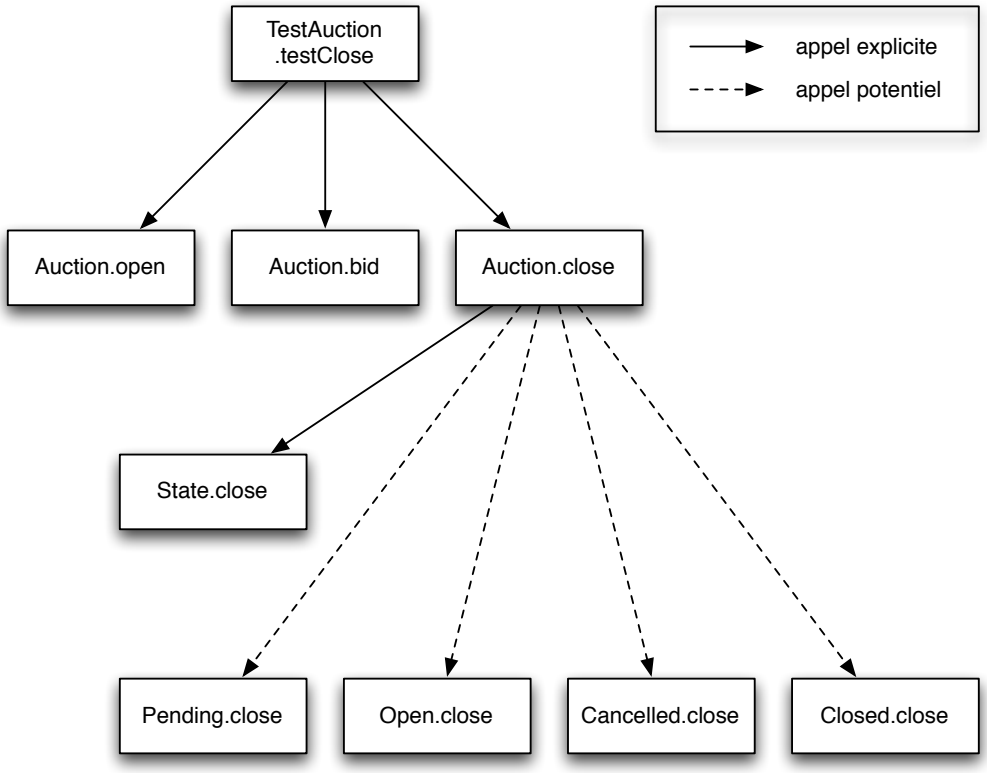

FIG. 3.5 – Un extrait du graphe d'appels statique du Listing 3.2.

vérifie (avec les assertions des lignes 9 et 10) si la vente est effectivement fermée et si l'acheteur est bien le meilleur enchérisseur. La Figure 3.5 montre un extrait du graphe d'appels statique de ce cas de test.

Vidock utilise Spoon, un outil d'analyse statique de programmes Java, pour construire le graphe d'appels statique. Spoon permet de visiter l'arbre syntaxique abstrait des sources Java à l'aide du patron de conception «visiteur». Un graphe d'appels statique sans appels potentiels peut être extrait directement d'un arbre syntaxique abstrait. L'arbre syntaxique abstrait d'une méthode détaille quels sont les appels explicites qui sont effectués. Nous pouvons ainsi obtenir rapidement une version du graphe d'appels statique d'un cas de test sans les appels potentiels.

Obtenir les appels potentiels nécessite d'explorer l'arbre syntaxique abstrait du système complet afin d'extraire toutes les méthodes redéfinissant la méthode appelée, et ce pour chaque appel. Les appels potentiels sont aussi extraits à l'aide de Spoon. Un arbre d'héritage est créé pour accéder directement aux sous-classes d'une classe. Puis, pour chaque appel explicite, nous vérifions si des sous-classes de la classe de la méthode appelée redéfinissent la méthode appelée. Pour chaque redéfinition trouvée, un appel potentiel est créé, avec la même source que l'appel explicite, et comme cible la redéfinition.

Cas de test impacté Un cas de test impacté est un cas de test qui peut potentiellement exécuter une méthode impactée. Soit t un cas de test, S son graphe d'appels statique, T_I l'ensemble des cas de test impactés et M_I l'ensemble des méthodes impactées, alors :

$$t \in T_I \Leftrightarrow \exists P = t, v_1, \ldots, v_n \text{ un chemin dans } S, v_n \in M_I$$

Selon la définition précédente, un cas de test est impacté si et seulement si, il est possible d'atteindre une méthode impactée à partir de la racine de son graphe d'appels statique. Donc une fois que son graphe d'appels statique a été construit, nous pouvons déterminer si un cas de test est impacté en utilisant l'algorithme du Listing 3.3. Cet algorithme retourne vrai si le nœud fournit en paramètre représente une méthode impactée ou peut atteindre un tel nœud. Si on exécute cet algorithme avec comme paramètre la racine d'un graphe d'appels statique d'un cas de test, on obtient vrai si le cas de test est impacté, faux sinon.

Cet algorithme implémente une recherche en profondeur d'abord sur un graphe et a donc une complexité en $\mathcal{O}(|V|+|E_a|+|E_p|)$, car V est l'ensemble des nœuds et $E_a \cup E_p$ est l'ensemble des arêtes. Comme $E_a \cap E_p = \emptyset$, $|E_a \cup E_p| = |E_a| + |E_p|$.

L'attribut `impactedMethods` représente l'ensemble des méthodes impactées. À la ligne 3, l'algorithme vérifie si le nœud courant représente une méthode impactée, et retourne vrai si c'est le cas.

L'attribut `v.next` représente l'ensemble $\{v' | (v, v') \in E_a \cup E_p\}$. La boucle de la ligne 7 appelle récursivement l'algorithme sur les nœuds qui peuvent être atteint en une transition, avec un appel explicite ou potentiel.

On dit qu'un nœud est sur un *chemin impacté* s'il est possible d'atteindre un nœud représentant une méthode impactée. L'attribut `onImpactedPath` représente un en-

```
1  boolean impacted(Vertex v) {
2    if onImpactedPath.contains(v) or
3       impactedMethods.contains(v.method) then
4      return true
5    end
6    beingVisited.add(v)
7    for each v' in v.next do
8      if not beingVisited.contains(v') then
9        if impacted(v') then
10         onImpactedPath.add(v')
11         beingVisited.remove(v)
12         return true
13       end
14     end
15   end
16   visited.remove(v)
17   return false
18 }
```

Listing 3.3 – L'algorithme déterminant si un cas de test est impacté.

semble de nœuds sur un chemin impacté. Cet attribut est utilisé pour optimiser l'algorithme lorsque plusieurs cas de test exécutent la même méthode. Si par une exécution précédente de l'algorithme on sait déjà qu'un nœud est sur un chemin impacté, on peut arrêter l'algorithme et retourner vrai. C'est le test qui est effectué à la ligne 2.

La notion d'appels potentiels entraîne une sur-approximation. Lorsqu'il y a des appels potentiels, certains ne sont pas réellement exécutés, mais statiquement on ne peut les distinguer. Si un cas de test est détecté comme impacté à cause d'un appel potentiel qui n'a jamais lieu on obtient alors un `faux positif`. De cette sur-approximation peut résulter des *faux positifs* (cas de test non-impactés détectés comme impactés) mais pas de *faux négatif* (cas de test impacté non détecté comme impacté).

Par exemple, sur la Figure 3.5, la méthode `Open.close` est impactée par l'aspect `Reserve`, le cas de test est donc détecté comme impacté. Si la méthode effectivement exécutée est `Open.close`, alors il s'agit d'un *vrai positif*, sinon il s'agit d'un *faux positif*.

3.2 Implémentation

Vidock a été implémenté comme une extension pour Eclipse, un environnement de développement populaire, disponible librement sur Internet [3].

Une fois l'extension installé dans Eclipse, l'analyse est automatiquement effectuée à chaque compilation d'un projet AspectJ. Le processus décrit Section 3.1 est appliqué,

[3] `http://www.irisa.fr/triskell/Softwares/protos/vidock/`

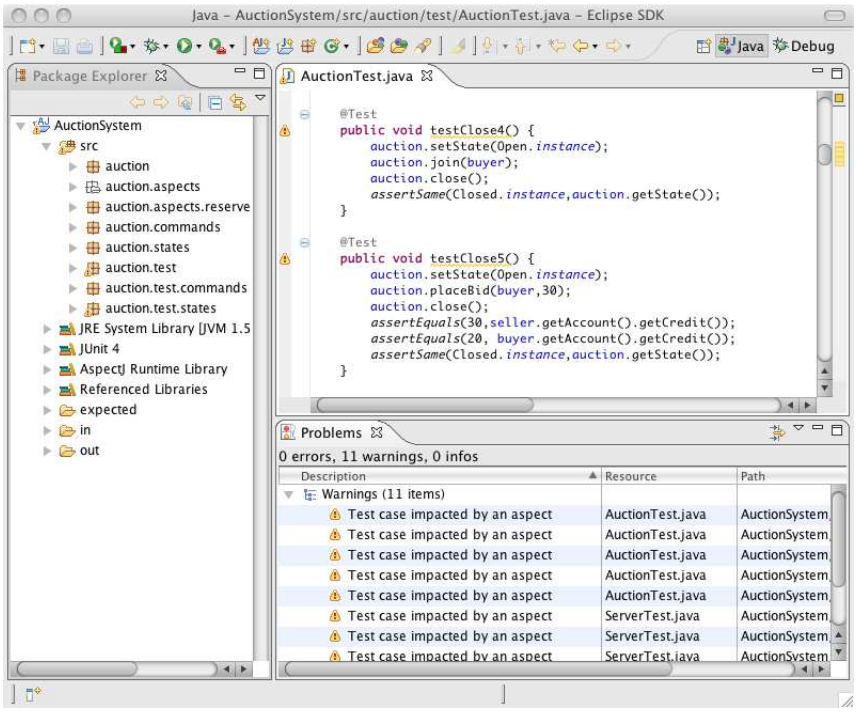

FIG. 3.6 – Capture d'écran de l'IDE Eclipse illustrant les avertissements indiqués par Vidock.

et pour chaque cas de test détecté comme impacté, un avertissement est rapporté, dans l'interface de gestion des erreurs et avertissements d'Eclipse.

À chaque démarrage d'Eclipse, l'extension enregistre un observateur, qui est appelé à chaque compilation d'un projet AspectJ. À cette étape, AJDT a déjà résolu les expressions de point de coupe et nous pouvons utiliser son interface de programmation pour obtenir les points de jonction où les greffons sont tissés. Puis nous utilisons l'interface de programmation d'Eclipse pour récupérer les classes du projet, et nous utilisons Spoon pour détecter les cas de test JUnit dans ces classes. Ensuite le processus est exécuté comme décrit dans la Section 3.1, and si un cas de test impacté est détecté, nous utilisons Spoon pour rapporter un avertissement.

La Figure 3.6 montre une capture d'écran de l'IDE Eclipse qui illustre comment Vidock averti des cas de test impactés. Pour chaque cas de test impactés détecté par Vidock, il y a un avertissement de Eclipse. Une icône d'avertissement s'affiche à côté du cas de test, et les cas de test impactés sont listés dans la liste des erreurs et avertissements d'Eclipse.

Afin d'optimiser le temps d'exécution, nous construisons les graphes d'appels statiques *à la volée*. Dans l'algorithme du Listing 3.3, l'ensemble `v.next` est construit dynamiquement à l'exécution. Ainsi, si un nœud représentant une méthode impactée est rencontré assez tôt, on évite de construire le graphe d'appels statique complet. Si un cas de test n'est pas impacté, son graphe d'appels statique est construit entièrement.

Pour chaque méthode du système il n'existe qu'un seul nœud de graphe d'appels statique en mémoire. Cela permet à la fois de limiter la consommation de mémoire et d'optimiser l'algorithme avec l'heuristique des *chemins impactés* (voir Section 3.1.2, page 68). Ainsi lorsque l'on rencontre pou la seconde fois un même nœud qui est sur un chemin impacté, on peut arrêter l'algorithme plus tôt.

Nous envisageons plusieurs améliorations pour l'outil Vidock. Il peut être utile de n'effectuer l'analyse à la demande de l'utilisateur, avec des options supplémentaires. Il peut par exemple être intéressant de connaître les raisons pour lesquels un cas de test impactés (c'est-à-dire, quels méthodes impactées sont exécutées par le cas de test, et quels aspects impactent ces méthodes). Cela implique de ne pas utiliser d'heuristiques et donc un temps d'exécution plus long, d'où l'intérêt de n'effectuer cette analyse plus poussée qu'à la demande de l'utilisateur.

3.3 Cas d'étude

Pour valider notre approche nous avons expérimenté notre analyse sur plusieurs études de cas. Nous avons utilisé des exemples distribués avec AspectJ (`Introduction`, `Bean`, et `Telecom`), `Tetris` qui est disponible librement sur Internet, et un projet de ventes aux enchères développé dans l'équipe Triskell (`Auction`).

L'objectif des ces études de cas et de montrer que l'approche proposée et implémentée par Vicock permet de détecter les cas de test impactés avec précision. Nous voulons aussi montrer que les sur-approximations, décrites dans la Section 3.1, induites par les expressions de point de coupes dynamiques et le polymorphisme, ont un impact négligeable sur les résultats de l'analyse. Enfin nous voulons observer la proportion et le nombre de cas de test impacté dans chaque projet. S'il y a peu de cas de test impactés, l'approche consistant a tester d'abord le programme de base seul est possible et permet un gain en ne ré-exécutant pas tous les cas de test.

3.3.1 Description

L'exemple `Introduction` a une unique classe `Point` qui représente les coordonnées d'un point dans un espace à deux dimensions, en utilisant soit un système cartésien, soit un système polaire). Trois aspects sont fournis, `Cloneable`, `Comparable`, et `Hashable`. Ces trois aspects ne possèdent pas de greffon. Chacun déclare que `Point` implémente une nouvelle interface, et ajoute les méthodes nécessaires à cette implémentation.

L'exemple `Bean` a aussi une classe `Point` représentant un point, mais seul le système cartésien est utilisé pour les coordonnées. Le seul aspect présent ajoute le patron de conception observateur à la classe `Point`. Des méthodes pour ajouter et retirer les observateurs sont ajoutées à la classe `Point`. Des greffons sont tissés pour notifier les observateurs des changements d'états.

L'exemple `Telecom` simule une communication téléphonique et gère les communications locales et longue distance, ainsi que les conversations avec plus de deux in-

TAB. 3.1 – Résultats des expériences.

Cas de test	Aspect	Cas de test sélectionnés	Cas de test réellement impactés	Précision	Rappel
Introduction					
7	Cloneable	0	0	NA	NA
	Comparable	0	0	NA	NA
	Hashable	0	0	NA	NA
Bean					
7	BoundPoint	6 (85,7 %)	6	100 %	100 %
Telecom					
33	Timing	3 (9,1 %)	3	100 %	100 %
	Billing	3 (9,1 %)	3	100 %	100 %
Tetris					
45	TestAspect	8 (17,8 %)	8	100 %	100 %
	NewBlocks	7 (15,6 %)	7	100 %	100 %
	Counters	2 (4,4 %)	2	100 %	100 %
	Levels	2 (4,4 %)	2	100 %	100 %
	All aspects	17 (37,8 %)	17	100 %	100 %
Auction					
306	Reserve	11 (3,6 %)	7	63,6 %	100 %
	AltBid	49 (16,0 %)	39	79,6 %	100 %
	Log	306 (100 %)	306	100 %	100 %
	Reserve + Altbid	49 (16,0 %)	39	79,6 %	100 %

terlocuteurs. L'aspect `Timing` calcule la durée d'une connexion et le temps total de connexion d'un consommateur. L'aspect `Billing` s'appuie sur `Timing` pour calculer le coût d'une connexion, selon qu'il s'agit d'un appel local ou longue distance.

Le projet `Tetris` est une implémentation en AspectJ du jeu vidéo classique, développée par Gustav Evertsson. L'aspect `TestAspect` trace les images utilisées et affiche leurs noms dans la console. L'aspect `NewBlocks` ajoute de nouveaux types de blocs au jeu. L'aspect `Counters` compte le nombre de lignes supprimées et affiche le total dans l'interface graphique du jeu. L'aspect `Levels` ajoute un système de niveaux au jeu : à chaque fois qu'un certain nombre de lignes supprimées a été atteint, le niveau est incrémenté et le jeu s'accélère.

Finalement, le projet `Auction` est le système de ventes aux enchères décrit dans la Section 1.1.2.

3.3.2 Résultats

Pour chaque expérience, nous avons écrit des cas de test avec la couverture d'instructions comme critère minimum. Ces cas de test ont permis de découvrir et corriger

plusieurs erreurs. Dans le cas du système de ventes aux enchères, ce sont les cas de test qui ont servi à mettre au point le système. Dans l'exemple `Introduction` d'AspectJ, nous avons découvert plusieurs erreurs dans la gestion des coordonnées polaires.

Ensuite les aspects ont été ajoutés aux programmes de base. Dans un premier temps chaque aspect est ajouté seul, ou avec les aspects dont il dépend s'il y en a. Dans un second temps les aspects sont tous ajoutés au programme de base. Après chaque tissage, l'analyse d'impact est effectuée sur les cas de test. La table 3.1 présente les résultats. D'abord, nous discutons les résultats particuliers (aucun ou presque tous les cas de test impactés), puis nous discutons des résultats normaux.

Dans le cas de l'exemple `Introduction`, aucun des cas de test n'est impacté par les aspects. En effet, aucun greffon n'est défini dans les trois aspects de cet exemple, et donc aucune méthode n'est impactée. Le code exécuté par les cas de test est donc le même, avec ou sans les aspects.

L'aspect `Log` du système `Auction` impacte tous les cas de test. Il s'agit d'un cas particulier car le greffon est tissé à l'exécution de toutes les méthodes du système et donc impacte toutes les méthodes. L'étude du Chapitre 2 a montré que la plupart des greffons étaient tissés à peu d'endroit, l'aspect `Log` est donc un cas isolé, bien que souvent pris comme exemple d'aspect transverse. .

Quand le système a très peu de classes, il est très probable qu'un aspect impacte une grande partie des cas de test. S'il y a peu de méthodes, il suffit qu'une méthode exécutée (ou potentiellement exécutée) dans tous les cas de test soit impactée pour que la majoritée des cas de test soit impactés. C'est le cas dans l'exemple `Bean`, où l'aspect `BoundPoint` impacte six cas de test sur sept. Ce genre de cas particulier a peu de chance de se produire dans un système plus grand.

Dans tous les autres cas, moins de 20 % des cas de test sont impactés, et dans certains cas, moins de cinq pour cent des cas de test sont impactés. Ces résultats montrent que notre analyse aide le programmeur à faire évoluer les cas de test. En effet, il peut se concentrer sur le sous-ensemble des cas de test impactés, sans avoir à faire la sélection manuellement. Cette analyse permet aussi, dans les exemples étudiés, de réduire le temps d'exécution des cas de test en évitant d'exécuter de 80 % à 95 % des cas de test.

Pour estimer les effets des sur-approximations induites par notre approche, nous avons calculé la précision et le rappel pour chaque aspect. La précision P et le rappel R sont définis en fonction des *vrais positifs* (vp, cas de test impactés détecté comme impactés), des *faux positifs* (fp, cas de test non-impactés détecté comme impactés), et des *faux négatifs* (fn, cas de test impactés non-détecté comme impactés) :

$$P = \frac{vp}{vp + fp}$$
$$R = \frac{vp}{vp + fn}$$

Pour calculer le rappel et la précision nous avons manuellement vérifié si chaque cas de test était impacté. Le résultat est présenté sur le tableau 3.1. Le rappel est toujours de 100 %, ce qui signifie qu'il n'y a jamais de *faux néfatif*, c'est-à-dire que notre approche

a permis de détecter tous les cas de test impactés sans exception. Dans la plupart des cas la précision est de 100 %, ce qui signifie qu'il n'y a pas de *faux positif*, tous les cas de test sélectionnés sont impactés. Avec l'aspect `Reserve`, la précision est de seulement 63,6 % mais il y a seulement quatre *faux positifs*. Dans les deux autres cas où la précision n'est pas de 100 %, la précision est bonne (79,6 %). Ces résultats montrent que la sur-approximation a un effet négligeable sur les résultats.

Lorsque plusieurs aspects sont ajoutés simultanément, le nombre de cas de test impactés n'est pas toujours égal à la somme des cas de test impactés par chaque aspect. Dans l'exemple `Introduction` aucun cas de test n'est impacté, le fait de tisser les aspects seuls ou ensemble ne fait donc pas de différence. Dnas le cas de `Bean`, il n'y a qu'un aspect. Dans le cas de `Telecom`, `Billing` dépend de `Timing`, les deux aspects sont donc tissés lorsque `Billing` est tissé. Dans le cas de `Tetris` le nombre de cas de test impactés par tous les aspects (17) est juste inférieur à la somme des cas de test impactés individuellement par chaque aspect (19). En revanche dans le cas de `Auction`, tous les cas de test impactés par `Reserve` sont aussi impactés par `AltBid`.

3.4 Conclusion

Afin d'aider le développeur à repérer les cas de test qui nécessitent d'être modifiés lors de l'introduction d'aspects, nous avons proposé une approche, implémentée dans un outil, Vidock. Cette approche statique détecte à la compilation les cas de test impactés, qui sont susceptibles d'être affectés par les changements introduits par les aspects.

Nous avons montré, à l'aide d'études de cas, que Vidock permettait de détecter avec une bonne précision les cas de test impactés. Ces études de cas ont aussi montré qu'il y a peu de cas de test impactés dans les exemples étudiés. Il est donc tout à fait envisageable de tester le programme de base sans les aspects afin d'améliorer la localisation des fautes, car le surcoût est faible.

Pour assurer qu'aucun cas de test impactés n'est ignoré, il est nécessaire d'effectuer des sur-approximations dans le cas des expressions de point de coupes dynamiques et du polymorphisme. Ces sur-approximations permettent de détecter tous les cas de test, comme montré dans nos études où le rappel est toujours de 100 %. Ces sur-approximations sont peu coûteuses car la précision est très bonne (100 % dans la majorité des cas).

Dans le futur, nous envisageons d'utiliser une classification des greffons, telle que celles proposées par Rinard *et al.* [Rinard 04] ou Munoz *et al.* [Munoz 08]. Selon l'impact d'un greffon sur le comportement du programme de base il est possible d'estimer le résultat attendu des cas des test impactés, ainsi que la façon dont ils doivent évoluer. Par exemple, certains greffons ne font qu'ajouter du comportement, sans changer l'existant. Dans ce cas on s'attend à ce que tous les cas de test impactés passent, et il faut augmenter l'oracle des cas de test afin de vérifier le nouveau comportement.

Chapitre 4

Test de l'expression de point de coupe

Dans un programme orienté-aspects, une faute peut être localisée dans le code de base, dans un greffon, ou dans une expression de point de coupe.

L'expression de point de coupe est la plus grande source d'erreur dans un programme orienté-aspects, comme observé par Ferrari *et al.* [Ferrari 08]. Dans la plupart des langages, les expressions de point de coupe sont définies de manière déclarative, ce qui introduit *la fragilité de l'expression de point de coupe*.

Il est donc nécessaire de développer de nouvelles techniques pour tester les expressions de point de coupe. Les fautes localisées dans le code de base ne sont pas spécifiques à la programmation orientée aspect. Même si les greffons sont spécifiques à la programmation orientée aspect, ils sont souvent écrits dans le même langage que le programme de base. Même s'il est nécessaire des les adapter, les techniques traditionnelles peuvent être utiles pour tester les greffons. Les expressions de point de coupe sont spécifiques à la programmation orientée aspect et utilisent souvent un langage qui leur est propre.

Lorsqu'on teste l'expression de point de coupe, on distingue trois types de fautes, définies par Lemos *et al.* [Lemos 06]. La Figure 4.1 illustre ces types de fautes, à l'aide d'ensembles. L'ensemble rayé correspond à l'ensemble des points de jonction attendus (c'est-à-dire qui correspond à l'intention du développeur), tandis que l'ensemble grisé correspond à l'ensemble des points de jonction effectivement sélectionnés par l'expression de point de coupe. Une faute dans l'expression de point de coupe peut produire à la fois des points de jonction non-attendus et des points de jonction négligés (1), seulement des points de jonction négligés (2), ou seulement des points de jonction non-attendus (3). Les fautes de type 1 peuvent être séparées en deux sous-types, selon que l'expression de point de coupe sélectionne des points de jonction attendus (1.a) ou non (1.b).

Dans ce chapitre nous proposons une approche, supportée par un outil nommé AdviceTracer, pour tester les expressions de point de coupe. Les greffons sont surveillés, afin d'enregistrer chaque exécution d'un greffon ainsi que le point de jonction où il s'est

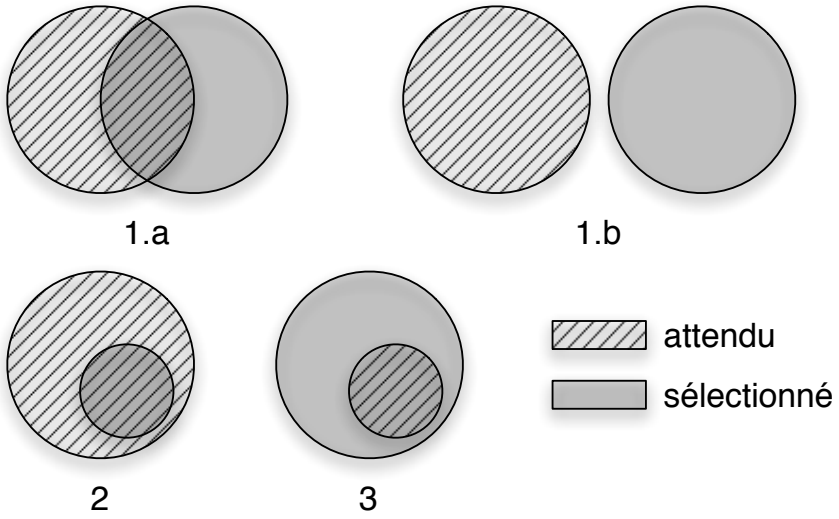

FIG. 4.1 – Les types de fautes possibles dans une expression de point de coupe.

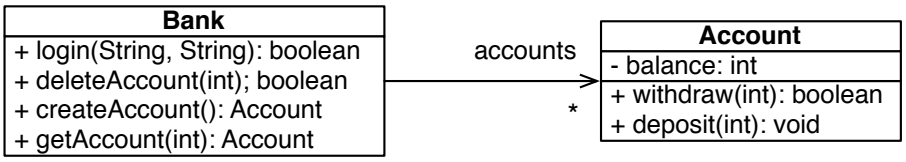

FIG. 4.2 – Diagramme de classe d'un système bancaire simplifié.

exécuté. Ces informations peuvent ensuite être utilisées par les cas de test pour vérifier qu'un greffon a été correctement tissé à tous les points de jonction attendus, ou pour vérifier que le greffon ne s'est pas exécuté depuis un point de jonction non-attendu.

La Section 4.1 détaille les problèmes et défis pour tester les expressions de point de coupe. La Section 4.2 présente l'approche proposée et AdviceTracer, notre outil. Dans la Section 4.3 nous présentons une étude empirique que nous avons menée afin de comparer des cas de test écrits avec AdviceTracer avec des cas de test écrits avec JUnit seulement, pour un système de taille moyenne. Enfin, la Section 4.4 conclut ce chapitre.

4.1 Défis pour le test d'expressions de point de coupe

4.1.1 Exemple de la banque

Afin d'illustrer sur un exemple simple les problèmes que l'on peut rencontrer lors du test d'expressions de point de coupe, nous utilisons l'exemple d'un système bancaire, dont le diagramme de classes est illustré sur la Figure 4.2. La classe `Bank` a des méthodes pour se connecter (`login`), créer ou supprimer un compte (`createAccount`

```
1  public aspect AccessControl {
2     pointcut controlledAccess():
3        execution(* Account.*(int));
4
5     @AdviceName("AccessControl")
6     before(): controlledAccess() {
7        if(!checkAccess(thisJoinPoint.getTarget()))
8           throw new DeniedAccessException();
9     }
10 }
```

Listing 4.1 – Un exemple d'aspect AspectJ pour le système bancaire.

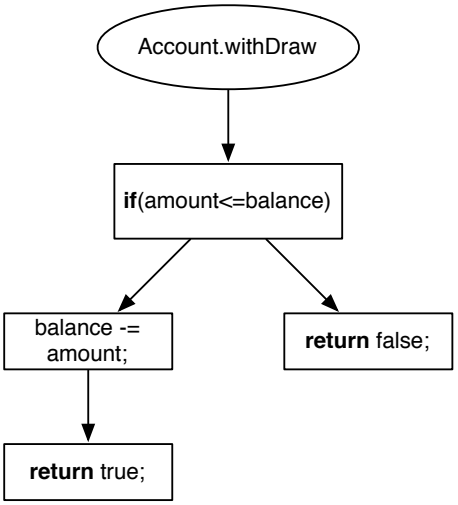

FIG. 4.3 – Graphe de flot de contrôle de la méthode `Account.withdraw`

et `deleteAccount`), et pour récupérer un compte à partir de son numéro (`getAccount`). La classe `Account` possède une méthode pour créditer (`deposit`) ou débiter (`withdraw`) le compte.

L'aspect du Listing 4.1 implémente une politique d'accès pour le système bancaire, afin d'assurer que seules les personnes autorisées peuvent accéder à un compte. Le greffon est tissé à l'exécution des méthodes de la classe `Account` ; si l'accès est autorisé alors la méthode s'exécute normalement, sinon une exception est levée.

4.1.2 Défis pour le test d'expressions de point de coupe

Une expression de point de coupe contenant une faute peut avoir de dangereux effets. La Figure 4.3 montre le graphe de flot de contrôle de la méthode `Account.withdraw` avant tissage, tandis que la Figure 4.4 montre le graphe de flot de contrôle après

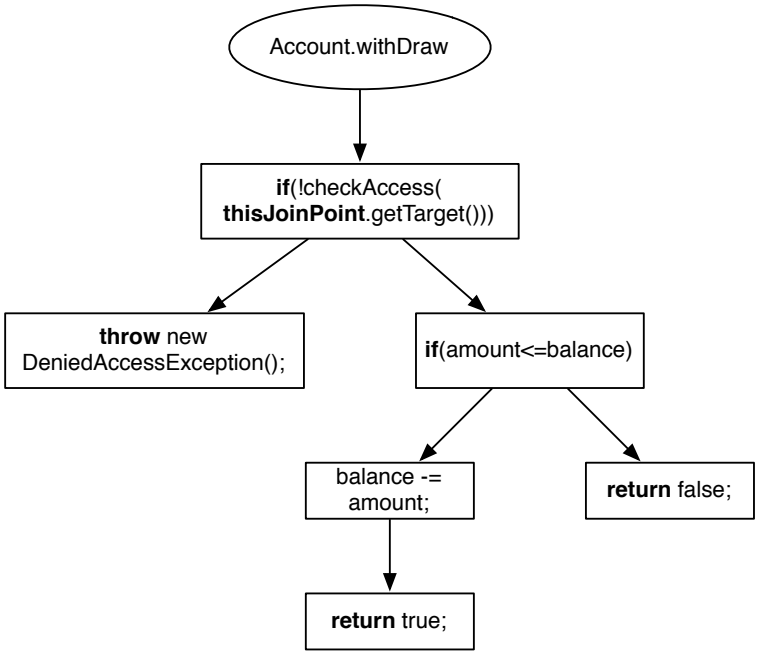

FIG. 4.4 – Graphe de flot de contrôle de la méthode `Account.withdraw` après tissage de l'aspect `AccessControl`

tissage de l'aspect `AccessControl`. L'aspect `AccesControl` modifie le flot de contrôle. Une instruction *if* a été ajoutée au début du flot de contrôle. Si la condition est satisfaite une exception est levée, sinon le flot de contrôle original est exécuté. Dans certains cas, le comportement original n'est donc pas exécuté. Une expression de point de coupe incorrecte peut introduire des problèmes critiques. Si l'expression de point de coupe sélectionne des points de jonction non-attendus, alors l'accès à certaines fonctions risque d'être bloqué par le greffon. Pire, si l'expression de point de coupe ne sélectionne pas certains points de jonction attendus, une faille de sécurité est introduit, car l'accès à un compte n'est plus sécurisé.

Pour tester une expression de point de coupe en AspectJ, il est actuellement possible

```
1 public void testAccessControl() {
2    bank.login(nonAuthorizedUser,password);
3    try {
4       account.withdraw(30);
5       fail("Access␣should␣not␣be␣authorized");
6    } catch(DeniedAccessException() {}
7 }
```

Listing 4.2 – Un cas de test pour l'aspect `AccessControl`.

d'utiliser JUnit pour la création et l'exécution des cas de test. Cependant, JUnit n'est pas complètement adapté au test de ces expressions. Le cas de test du Listing 4.2 illustre les défis pour tester une expression de point de coupe en utilisant JUnit seul. Pour vérifier que l'expression de point de coupe est correcte, il faut vérifier que le greffon qui lui est associé a été tissé aux bons endroits. Une façon de le faire est de vérifier que le comportement du greffon s'est exécuté correctement aux points où il est attendu. Le cas de test du Listing 4.2 connecte un utilisateur qui n'est pas autorisé à retirer de l'argent sur un compte particulier, et qui tente tout de même de retirer de l'argent sur ce compte. Si une exception est levée, le cas de test se termine et réussi, mais si aucune exception n'est levée, le cas de test échoue (l'assertion de la ligne 5 échoue toujours si elle est exécutée).

Ce cas de test est simple et peut sembler une bonne manière de tester une expression de point de coupe, mais il y au moins deux problèmes avec ce cas de test. Le premier problème est que si le cas de test échoue, on ne peut pas conclure qu'il y a une faute dans l'expression de point de coupe :

- Si le premier *joker* «`*`» à la ligne 3 du Listing 4.1 est remplacé par «`void`», la méthode `withdraw` n'est plus sélectionnée, et `testAccessControl` échoue car il y a une faute dans l'expression de point de coupe.
- Si à la ligne 7 du Listing 4.1, la négation («`!`») est retirée, le comportement du greffon est inversé, et `testAccessControl` échoue. Pourtant dans ce cas, l'expression de point de coupe est correcte est la faute détectée se trouve dans le greffon.

Le second problème est que `testAccessControl` ne peut pas détecter toutes les fautes dans l'expression de point de coupe. Par exemple, supposons que dans le motif de méthode du Listing 4.1, ligne 3, le nom `Account` soit remplacé par un joker. Le greffon est alors tissé avant les exécutions de la méthode `Bank.deleteAccount`, mais `testAccessControl` réussi même s'il y a une faute dans l'expression de point de coupe. Il s'agit d'une faute de type 3 comme définit sur la Figure 4.1, ce qui veut dire que les points de jonction attendus sont sélectionnés, mais que l'expression de point de coupe sélectionne aussi des points de jonction non-attendus. Pour détecter une telle faute, il faut écrire des cas de test qui spécifient qu'aucun greffon n'est tissé à différents endroits du code. Un cas de test tel que `testAccessControl` ne peut pas vérifier cela, car il ne mentionne pas explicitement la présence ou l'absence d'un greffon, il présume implicitement sa présence. Il s'agit d'une limitation de JUnit qui ne permet pas de définir des assertions spécifiant explicitement la présence d'un greffon.

Pour résoudre ces problèmes, nous avons besoin d'une façon de définir un oracle qui cible spécifiquement les fautes dans les expressions de point de coupe, et qui ne repose pas sur le comportement du greffon. Dans la section suivante nous proposons une solution pour la définition d'oracles dédiées aux expressions de point de coupe. Cette solution est basée sur la surveillance des exécutions des greffons lorsque les cas de test s'exécutent, et sur la définition d'un ensemble d'assertions dédiées, pour effectuer des requêtes sur ces exécutions. Cette solution a été implémentée dans un outil nommé AdviceTracer et les assertions peuvent être utilisées pour définir un oracle automatique

pour les expressions de point de coupe.

4.2 AdviceTracer

L'outil AdviceTracer [1] permet au développeur d'écrire des cas de test qui se concentrent sur l'expression de point de coupe. Plus précisement AdviceTracer est utilisé pour spécifier des oracles qui vérifient la présence ou l'absence d'un greffon dans le flot d'exécution.

4.2.1 Approche

AdviceTracer implémente une approche pour tester les expressions de point de coupe. Le principe est que les exécutions des greffons durant le scénario de test sont enregistrées et que des assertions sont exprimées sur ces traces d'exécutions au lieu du comportement du greffon.

L'approche est séparée en deux étapes, la première pour vérifier que tous les points de jonction attendus ont été sélectionnés, et la seconde pour vérifier qu'aucun point de jonction non-attendu n'a été sélectionné.

Dans la suite de cette section, nous décrivons comment les cas de test sont écrits pour les points de jonction attendus et non-attendus. Chaque test est composé d'un scénario de test et d'un oracle (lui-même composé d'assertions). Nous discutons ensuite la façon d'interpréter le résultat des cas de test pour localiser les fautes.

Cas de test pour les points de jonction attendus Dans la première étape, les cas de test sont écrits pour vérifier que tous les points de jonction attendus ont été sélectionnés par les expressions de point de coupe. Il est nécessaire d'écrire un cas de test pour chaque point de jonction attendu. Cela peut sembler coûteux, mais l'étude du Chapitre 2 montre que la plupart des greffons n'est tissé qu'à très peu de points de jonction. Il n'y a donc souvent que quelques cas de test à écrire pour chaque expression de point de coupe.

Le scénario de test d'un cas de test qui cible les points de jonction attendus doit être le plus simple possible. L'unique but du scénario de test est d'exécuter un point de jonction attendu en particulier.

L'oracle doit vérifier que le greffon associé à l'expression de point de coupe testée a bien été exécuté, et qu'il a bien été exécuté au bon endroit. Les assertions doivent donc être spécifiques pour s'assurer que le point de jonction attendu a bien été sélectionné par l'expression de point de coupe.

Cas de test pour les points de jonction non-attendus Dans la seconde étape, les cas de test sont écrits pour vérifier qu'aucun point de jonction non-attendu n'a été sélectionné par les expressions de point de coupe. Les points de jonction non-attendus

[1] `http://www.irisa.fr/triskell/Softwares/protos/advicetracer/`

peuvent se trouver n'importe où dans le programme, il peut donc être difficile d'écrire de tels cas de test. Notre approche permet de réduire le coût de rédaction des cas de test pour les points de jonction non-attendus.

Afin de limiter le nombre de cas de test pour valider l'absence de point de jonction non-attendu, il est nécessaire d'écrire des cas de test qui couvrent de larges portions du programme. Comme AdviceTracer est basé sur les traces d'exécutions, plus la trace d'exécutions est grande, plus le nombre de vérifications possibles avec un cas de test est grand. AdviceTracer offre une assertion qui vérifie le nombre d'occurrence d'un greffon spécifique dans la trace d'exécutions. Elle peut-être utilisée pour valider qu'un greffon n'a pas été tissé plus de fois qu'attendu dans une grande trace d'exécutions.

Il peut sembler difficile d'interpréter un résultat simplement basé sur un nombre d'occurrences. En effet, si le nombre d'exécutions d'un greffon est conforme à ce qu'on attend, on peut toujours se demander si ces exécutions ont eu lieu aux bons points de jonction. C'est pourquoi il est important de tester les points de jonction attendus en premier. Si on est sûr que les points de jonction attendus sont correctement sélectionnés, et si il y a autant d'exécutions d'un greffon qu'attendu, on est sûr qu'il n'y a pas de point de jonction non-attendu sélectionné. De cette façon, l'oracle reste simple, même si le scénario de test est complexe.

Localisation des fautes Si une faute est détectée, il est possible de savoir de quel type de faute il s'agit et où elle est localisée, en interprétant les résultats des cas de test.

D'abord, il faut regarder les résultats des cas de test pour les points de jonction attendus. Si un cas de test pour les points de jonctions attendus échoue, le point de jonction ciblé n'a pas été sélectionné par l'expression de point de coupe testée. La faute est donc simplement localisée car chaque cas de test ne cible qu'une expression de point de coupe et qu'un point de jonction.

Ensuite, s'il y a toujours des fautes, il faut regarder les résultats des cas de test pour les points de jonction non-attendus. Si aucun cas de test pour les points de jonction attendus n'échoue, et qu'un cas de test pour les points de jonction non-attendus échoue, au moins un point de jonction non-attendu a été sélectionné par les expressions de point de coupe. Comme chaque assertion ne teste qu'un greffon, il est possible de savoir quel greffon a été tissé à un point de jonction non-attendu, ce dernier se trouvant forcément dans le code couvert par le scénario de test.

Pour les points de jonction non-attendus, il est nécessaire de trouver un compromis entre des scénarios de test trop petits et des scénarios de test trop grands. Des scénarios de test petits améliorent la localisation des fautes, tandis que des scénarios de test grands réduisent le nombre de cas de test à écrire, et donc l'effort de test.

Pour résumer, si un cas de test pour les points de jonction attendus échoue, il y a une faute de type 1 ou type 2, et si un cas de test pour les points de jonction non-attendus échoue, il y a une faute de type 1 ou type 3.

```
1 public class C {
2   public void m1() { ... }
3   public void m2() {
4     ...
5     m1();
6   }
7 }
```

Listing 4.3 – A Java class example

```
1 public Aspect A {
2   @AdviceName("A1")
3   before(): execution(void C.m1())
4         && cflow(execution(void C.m2())) {
5     ...
6   }
7 }
```

Listing 4.4 – An AspectJ aspect example

```
1  public class Test {
2    @Test
3    public void test1() {
4      C c = new C();
5      addTracedAdvice("A1");
6      setAdviceTracerOn();
7      c.m1();
8      setAdviceTracerOff();
9      assertAdviceExecutionsEquals("A1",0);
10   }
11
12   @Test
13   public void test2() {
14     C c = new C();
15     addTracedAdvice("A1");
16     setAdviceTracerOn();
17     c.m2();
18     setAdviceTracerOff();
19     assertAdviceExecutionsEquals("A1",1);
20     assertExecutedAdviceAtJoinpoint("A1","C.m1:2");
21 }
```

Listing 4.5 – A JUnit test class illustrating how to use AdviceTracer

4.2.2 Exemple illustratif

Les Listings 4.3, 4.4, 4.5 illustrent l'utilisation d'AdviceTracer. Le Listing 4.3 montre une classe Java `C` avec deux méthodes `m1` et `m2` ; `m2` appelle `m1`. Le Listing 4.4 montre un aspect AspectJ `A`, avec un greffon `A1`. Ce greffon est tissé avant les exécutions de `m1` qui se trouvent dans le flot de contrôle d'une exécution de `m2`.

Pour faciliter la désignation des greffons, il est recommandé d'utiliser l'annotation Java 5 `@AdviceName`. Cette annotation, définie par AspectJ, permet de nommer un greffon en passant une chaîne de caractères comme paramètre, comme sur la ligne 2 du Listing 4.4.

Le Listing 4.5 montre deux cas de test qui spécifient l'ensemble des points de jonction attendus. Sous une forme textuelle, la spécification des points de jonction attendus peut-être exprimée comme ceci : « le greffon `A1` doit être tissé seulement avant les exécutions de `m1` ayant lieu dans le flot de contrôle de `m2` ». Nous devons donc spécifier que le greffon `A1` est attendu quand `m2` appelle `m1`, mais aussi que `A1` n'est pas attendu quand `m1` s'exécute en dehors du flot de contrôle de `m2` :

- `test1` appelle directement `m1`, et vérifie que le greffon `A1` n'est pas exécuté.
- `test2` appelle `m2`, et vérifie que le greffon `A1` est exécuté depuis la ligne 2 de `C`, dans `m1`.

Si l'expression de point de coupe du Listing 4.4 est remplacée par «**`execution`** `(`**`void`** `C.m1())`», alors `test1` échoue car le greffon est exécuté dans ce scénario de test. Si l'expression de point de coupe est remplacée par «**`execution`**`(`**`void`** `C.m2())`», alors `test2` échouera car le greffon sera tissé dans `m2` au lieu de `m1`.

Cet exemple illustre aussi comment AdviceTracer permet de gérer les expressions de point de coupe dynamiques. L'expression de point de coupe du Listing 4.4 est dynamique : il sélectionne les exécutions de `m1` dans le flot de contrôle de `m2`. Pour spécifier cette expression de point de coupe nous exécutons d'abord `m1` hors du flot de contrôle de `m2` pour vérifier que le greffon n'est pas exécuté (`test1`), puis nous exécutons `m1` dans le flot de contrôle de `m2` pour vérifier que le greffon est bien exécuté au bon endroit (`test2`).

AdviceTracer permet de spécifier les points de jonction attendus pour vérifier qu'il n'y a pas de point de jonction négligé. Pour cela, il faut écrire des cas de test qui spécifient tous les points de jonction qui doit être sélectionné par chaque expression de point de coupe. AdviceTracer permet aussi d'écrire des cas de test qui vérifient qu'il n'y a pas de point de jonction non-attendu.

4.2.3 Méthodes primitives d'AdviceTracer

Les deux cas de test du Listing 4.5 illustrent comment les méthodes primitives d'AdviceTracer sont utilisées. Ces primitives sont de quatre types différents : celles pour démarrer ou arrêter AdviceTracer, celles pour récupérer les traces d'exécutions des greffons, celles qui configurent les greffons tracés, et les assertions pour spécifier l'oracle.

Démarrer et arrêter AdviceTracer

Par défaut, AdviceTracer n'enregistre aucune trace. Pour faire des assertions sur le nombre d'exécutions des greffons, AdviceTracer doit savoir quand le scénario de test commence, et quand il finit. Dans chaque cas de test il est donc nécessaire de démarrer AdviceTracer juste avant le scénario de test et de l'arrêter juste après.

AdviceTracer est démarré en appelant la méthode `setAdviceTracerOn()`. La méthode `setAdviceTracerOff()` arrête AdviceTracer. Entre les exécutions de `setAdviceTracerOn` et `setAdviceTracerOff`, chaque exécution d'un greffon est enregistrée, ainsi que la position du point de jonction d'où il a été exécuté.

Restreindre les greffons observés

Par défaut AdviceTracer observe tous les greffons du système. Il est possible de spécifier les greffons observés à l'aide des méthodes `setTracedAdvices` et `addTracedAdvice`. La première méthode prend en paramètre une liste de noms de greffons et spécifie l'ensemble exacte des greffons observés. La méthode `addTracedAdvice` ajoute le greffon dont le nom est donné en paramètre à la liste des greffons observés.

Limiter le nombre de greffons observés permet de rendre les cas de test plus robuste aux évolutions. Ainsi l'oracle se limite à un petit nombre de greffons, et est donc insensible aux ajouts ou modifications des autres greffons. En particulier, les assertions sur le nombre de greffons exécutés peuvent être affectées par les autres greffons.

Assertions fournies par AdviceTracer

AdviceTracer fournit trois nouvelles assertions qui s'ajoutent aux assertions fournies par JUnit.

assertAdviceExecutionsEquals(String advice, int n) Cette assertion réussit si le nombre d'exécutions du greffon spécifié est égal au paramètre entier n. Cette assertion est requise pour détecter les points de jonction non-attendus, comme expliqué dans la Section 4.2.1.

assertExecutedAdvice(String advice) Cette assertion réussit si le greffon dont le nom est passé en paramètre (*advice*) a été exécuté.

assertExecutedAdviceAtJoinpoint(String advice, String joinpoint) Cette assertion réussit si le greffon dont le nom est passé en paramètre (*advice*) a été exécuté au point de jonction passé en paramètre. Le format du paramètre joinpoint est précisé dans la Section 4.2.4.

4.2.4 Implémentation d'AdviceTracer

AdviceTracer fournit les primitives décrites précédemment et contient un aspect, écrit en AspectJ, qui peut être tissé à la compilation ou lors de l'exécution. Un tissage à la compilation offre de meilleurs performances, mais un tissage à l'exécution permet d'exécuter les cas de test sans modifier le *bytecode*.

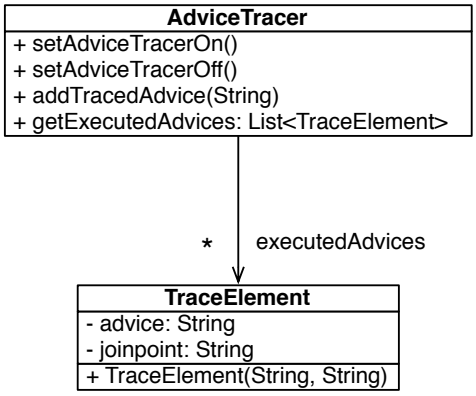

FIG. 4.5 – Le diagramme de classes d'AdviceTracer.

Les informations enregistrées par AdviceTracer peuvent être accédées avec un ensemble de méthodes statiques utilisées pour définir les cas de test. La Figure 4.5 montre le diagramme de classes d'AdviceTracer. Le greffon dans cet aspect récupère le nom du greffon, et la localisation du point de jonction d'où le greffon s'exécute. Cette information est stockée dans un élément de classe `TraceElement`, qui est une paire de chaînes de caractères. La chaîne de caractères pour le greffon est son nom, spécifié par l'annotation `@AdviceName`. La chaîne de caractères pour le point de jonction est construite avec le nom qualifié de la méthode et le numéro de ligne où le point de jonction se trouve (séparés par le caractère « `:` »).

Le greffon est tissé avant chaque exécution d'un autre greffon. L'expression de point de coupe du greffon est «**`adviceexecution`**`()&& !`**`within`**`(AdviceTracer)`».

La Figure 4.6 montre le diagramme de flot de contrôle de la méthode `Account.withdraw` après le tissage d'AdviceTracer. Il n'y a pas de branchement conditionnel dans le flot de contrôle ajouté, AdviceTracer ne modifie donc pas le comportement du programme de base et des aspects. Ici, lorsque la méthode `Account.withdraw` est exécutée, AdviceTracer est d'abord exécutée, puis le greffon `AccessControl` est exécuté.

Toutes les instances de `TraceElement` sont stockées dans une liste qui peut être récupérée en appelant la méthode statique `getExecutedAdvices`. Cette liste est remise à zéro à chaque démarrage d'AdviceTracer, et ne contient donc que les instances de `TraceElement` enregistrées entre les précédents appels à `setAdviceTracerOn` et `setAdviceTracerOff`.

4.3 Étude empirique

Le but de cette étude empirique est de déterminer si AdviceTracer améliore effectivement le test d'expressions de point de coupe. Pour cela, nous comparons deux suites de tests, une écrite seulement avec des assertions JUnit normales, et l'autre à l'aide

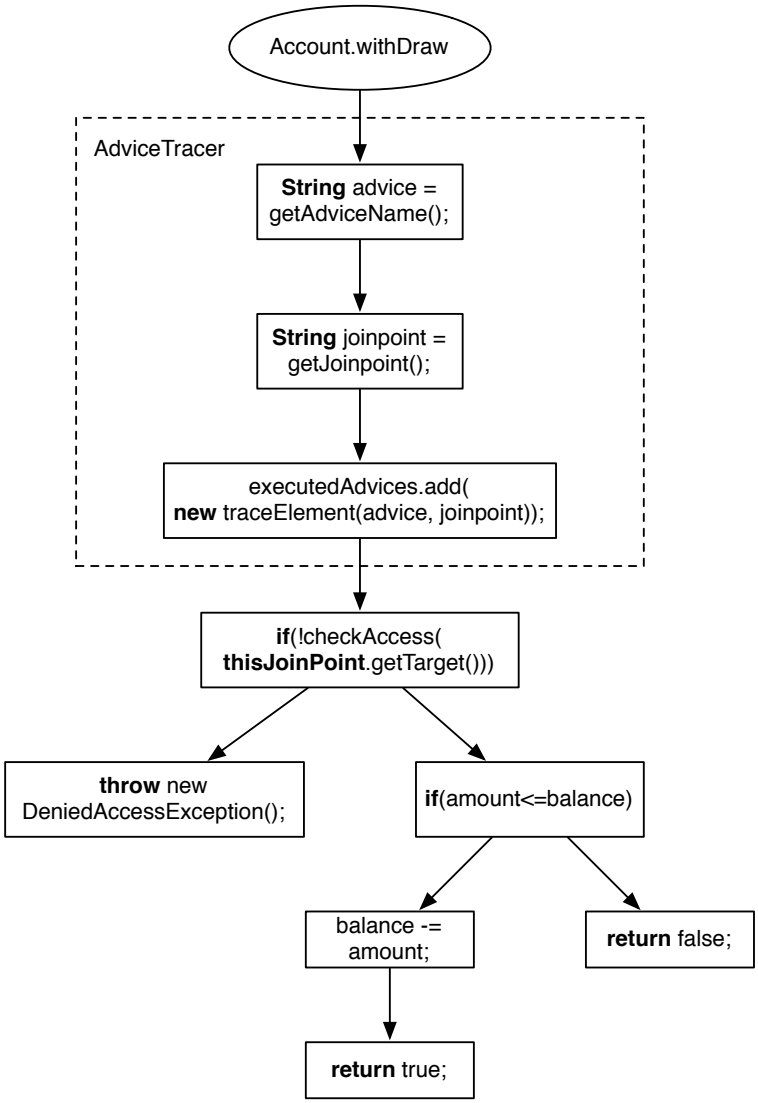

FIG. 4.6 – Le graphe de flot de contrôle de la méthode `Account.withdraw` après le tissage d'AdviceTracer.

d'AdviceTracer et JUnit. Les deux suites de tests spécifient les expressions de point de coupe du même système. Dans cette section, la suite de tests écrite seulement avec JUnit est appelée T_{JU}, tandis que la suite de tests écrite avec AdviceTracer est appelée T_{AT}.

4.3.1 Le système HealthWatcher

Le système testé est le système HealthWatcher. Il s'agit de la troisème version AspectJ du système étudié dans le Chapitre 2, décrit Section 2.1 page 39. Cette version implémente les patrons de conception «état» et «commande».

Nous avons légèrement modifié le système HealthWatcher. Dans le système HealthWatcher, il y a un mécanisme de persistance des données qui peut être désactivé. Le mécanisme de persistance repose sur une base de données fournies, qui n'est compatible qu'avec Microsoft Windows. Les développeurs ne donnant aucune information sur la base de données (tables, champs, etc.), il n'est pas possible de créer ou de remettre à zéro la base de données. Le seul moyen d'activer la persistance est donc d'utiliser Windows et la base de données fournie, ce qui veut dire que l'état de la base de données au début d'un cas de test est l'état de la base de données à la fin du cas de test précédent. Ceci peut introduire des effets de bord entre les cas de test et réduit la testabilité. Nous avons donc décidé de désactiver le mécanisme de persistance. Quatre aspects ont été supprimés, car ils n'avaient aucune utilité sans le mécanisme de persistance.

Le système HealthWatcher possède 93 classes, avec 530 méthodes, neuf aspects, et 19 greffons (et donc 19 expressions de point de coupe). Au total il y a 6435 lignes de code, dont 438 pour les aspects. La Table 4.1 montre le nombre d'occurrences de chaque expression de point de coupe primitive dans le système HealthWatcher.

4.3.2 Questions de recherche

Avec cette étude empirique, nous voulons réponde à deux questions de recherche.

Question 1 *AdviceTracer réduit il l'effort pour tester des expressions de point de coupe par rapport à JUnit ?* Tester des expressions de point de coupe en utilisant seulement JUnit peut être difficile ; AdviceTracer vise à réduire l'effort requis pour écrire les cas de test. Nous devons donc vérifier qu'AdviceTracer atteint effectivement ce but.

L'effort de test peut être évalué en comparant la taille des deux suites de tests, en termes de nombre de cas de test et en termes de nombre de lignes de code. Si une suite de tests détecte les même fautes avec moins de cas de test, cela veut dire que ses cas de test sont plus efficaces car ils détectent plus de fautes en moyenne. D'un autre côté, si une suite de tests détecte les même fautes avec des cas de test plus petits, il y a de fortes chances que ses cas de test soient plus facile à écrire. Il faut donc comparer le nombre de cas de test, le nombre total de lignes de code, et le nombre de lignes de code par cas de test moyen.

TAB. 4.1 – Nombre d'occurrences de chaque expression de point de coupe primitive PCD dans le système HealthWatcher.

Primitive PCD	**Number of Occurrences**
adviceexecution	0
args	12
call	2
cflow	0
cflowbelow	0
execution	10
get	0
handler	0
if	0
initialization	8
preinitialization	0
set	0
staticinitialization	0
target	12
this	1
within	3
withincode	0

Question 2 *AdviceTracer augmente-t-il la capacité des cas de test à détecter des fautes dans les expressions de point de coupe comparé à JUnit ?* Certaines fautes peuvent être difficiles ou impossibles à détecter en utilisant seulement des assertions JUnit. AdviceTracer cible spécifiquement les fautes dans les expressions de point de coupe, il faut donc vérifier qu'une suite de tests AdviceTracer permet de détecter au moins autant de fautes qu'une suite de tests JUnit. Comme nous avons utilisé l'analyse de mutation, nous avons considéré le score de mutation de chaque suite de tests comme indicateur du nombre de fautes détectées.

4.3.3 Protocole expérimental

Deux développeurs, expérimenté avec JUnit, ont développé les deux suites de tests en parallèle. Le premier a écrit des cas de test en n'utilisant que JUnit, tandis que le second a utilisé AdviceTracer. Aucun des deux développeurs n'était familier avec le système HealthWatcher. Les deux développeurs connaissaient JUnit, et le développeurs ayant utilisé AdviceTracer connaissait AdviceTracer. Les deux développeurs ont enregistré le temps passé à écrire les cas de test.

La génération des deux suites de tests devaient être dirigée par un critère de test commun. Nous avons utilisé le score de mutation comme critère de test. L'objectif était d'atteindre un score de mutation de 100 % si possible, ou le plus élevé possible sinon. Les expressions de point de coupe mutantes ont été générées à l'aide d'AjMutator, décrit dans le Chapitre 5.

AjMutator a généré 343 mutants. AjMutator est capable de détecter automatiquement une partie des mutants équivalents. Après compilation, 243 mutants étaient compilable et non détecté comme équivalent. Toutefois, il y avait encore des mutants équivalents non détectés par AjMutator.

Les deux développeurs devait trouver les mutants équivalents seul, sans se communiquer leurs résultats. Identifier les mutants équivalents peut demander beaucoup de temps, et les résultats seraient donc influencés si les développeurs partageaient leurs résultats. À la fin de l'étude, 76 mutants était identifiés comme non-équivalents.

4.3.4 Résultats

Question 1 La suite T_{JU} a demandé plus de temps à écrire. La suite T_{AT} a été écrite en approximativement 420 minutes (sept heures), tandis que T_{JU} a été écrite en approximativement 480 minutes (huit heures). La Figure 4.7 décrit l'évolution du score de mutation des deux suites de tests au cours du temps. Le score de mutation de T_{AT} est toujours plus élevé que le score de mutation de T_{JU}, ce qui veut dire qu'il est plus rapide de trouver des fautes avec AdviceTracer.

La Table 4.2 montre des métriques des deux suites de cas de test. Malgré un nombre de cas de test similaire (26 pour T_{JU}, 24 pour T_{AT}), T_{JU} a significativement plus de lignes de code que T_{AT} (respectivement 1 298 et 324 lignes de code).

Les cas de test écrit avec AdviceTracer sont plus petits que les cas de test écrit seulement avec JUnit. Les cas de test de T_{JU} font en moyenne 49,92 lignes de code, tandis que

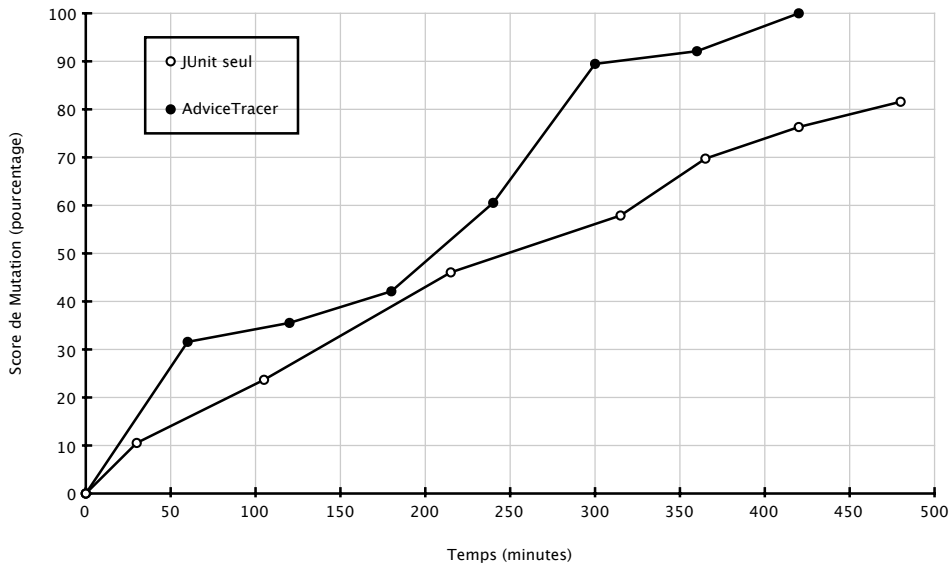

FIG. 4.7 – Évolution du score de mutation des deux suites de tests au cours du temps.

TAB. 4.2 – Métriques des deux suites de tests.

	JUnit seul	**AdviceTracer**
Cas de test	26	24
Lignes de code	1 298	324
Lignes de code par cas de test Moyenne	49,92	13,50
Lignes de code par cas de test Écart type	27,66	7,76
Time (minutes)	480	420

les cas de test de T_{AT} font en moyenne 13,50 lignes de code. L'écart type du nombre de lignes de code par cas de test est aussi plus faible avec AdviceTracer (7,76 contre 27,66 pour T_{JU}). Cela veut dire que les cas de test AdviceTracer ont des tailles très similaires (de 7 à 30 lignes de code) tandis que les cas de test JUnit ont des tailles très variables (de 19 à 137 lignes de code). Il n'y a aucune relation évidente entre une expression de point de coupe et la taille des cas de test qui la testent.

Afin d'illustrer pourquoi les cas de test JUnit sont plus grands que les cas de test AdviceTracer, nous analysons en détail le développement de cas de test pour une expression de point de coupe de HealthWatcher. Les Listings 4.6 et 4.7 montrent deux exemples de cas de test pour l'expression de point de coupe du greffon `AnimalComplaintStateAspect3` (Listing 4.8), dans l'aspect `AnimalComplaintStateAspect`. Ce greffon change l'état d'un objet `AnimalComplaint` lorsque la méthode `AnimalComplaintState.setStatus` est appelée.

En utilisant seulement JUnit (Listing 4.6), il faut vérifier que l'état est correctement changé lorsque `setStatus` est appelée. Le problème est que l'état est un attribut privé. La seule façon de vérifier que l'état a changé est d'exécuté une méthode dont le comportement dépend de l'état. C'est le cas de la méthode `setAnimal` qui change la valeur de l'attribut `animal` si la plainte est ouverte, mais ne fait rien si la plainte est fermée. Donc la première chose) faire est de créer un objet `AnimalComplaint` dans l'état ouvert (ligne 18). Ce constructeur requiert une liste de paramètres, qui sont initialisés de la ligne 3 à la ligne 16. La plainte est initialisée avec la valeur «`elephant`» pour l'attribut `animal`. À la ligne 27 la plainte est fermée. À la ligne 28 le cas de test essaye de passer la valeur de `animal` à «`tiger`». Si l'état a bien été changé par le greffon, la valeur de `animal` doit rester «`elephant`», comme vérifié à la ligne 30.

AdviceTracer offre un ensemble d'assertions pour vérifier la présence d'un greffon, qui permettent au testeur de cibler spécifiquement les fautes dans les expressions de point de coupe et réduit la complexité des cas de test. Comme il n'est pas nécessaire de tester le comportement exécuté, il est possible d'utiliser le constructeur vide pour instancier une plainte, à la ligne 3 du Listing 4.7. À la ligne 4, le cas de test spécifie que seul le greffon `AnimalComplaintStateAspect3` doit être surveillé. Puis à la ligne 5, AdviceTracer est démarré. La méthode `AnimalComplaint.setStatus` est appelé à la ligne 6 ; cette méthode appelle `AnimalComplaintState.setStatus` où le greffon doit être tissé. Une fois que le scénario de test est terminé, AdviceTracer est arrêté, à la ligne 7. Enfin l'oracle est composé de deux assertions, aux lignes 8 et 9. La première assertion vérifie que le greffon a bien été exécuté une seule fois, et la seconde vérifie que le greffon a été exécuté au bon point de jonction.

Cet exemple montre plusieurs avantages d'AdviceTracer. Le scénario de test est plus simple avec AdviceTracer. Comme il n'est pas nécéssaire de vérifier le comportement du greffon, les cas de test ont seulement besoin d'exécuter les points de jonction où le greffon est attendu. En n'utilisant que JUnit,il est nécessaire de mettre en œuvre un scénario de test complexe afin d'observer le comportement du greffon.

Les oracles écrits avec AdviceTracer sont plus précis. Le cas de test écrit avec AdviceTracer ne peut réussir que si l'expression de point de coupe sélectionne l'ensemble

```
1  public class JUnitTest extends TestCase {
2        public void testAnimalComplaintStateAspect3() {
3              Employee employee = new Employee("login",
4                                    "password", "name");
5              Date date1 = null;
6              Date date2 = null;
7              Date date3 = null;
8              try {
9                    date1 = new Date(21, 7, 2009);
10                   date2 = new Date(22, 7, 2009);
11                   date3 = new Date(23, 7, 2009);
12             }
13             catch(InvalidDateException e) {
14                   (*\textit{fail}*)("Invalid␣date");
15             }
16             Address address = new Address();
17
18             AnimalComplaint complaint = new AnimalComplaint(
19                "complainant", "description", "observation",
20                "e-mail", employee, open, date1, date2, address,
21                date3, "elephant", address);
22
23             AnimalComplaintState state = new
24                AnimalComplaintStateOpen();
25             state.setAnimal(complaint.getAnimal());
26
27             state.setStatus(closed, complaint);
28             complaint.setAnimal("tiger");
29
30             (*\textit{assertEquals}*)("elephant", complaint.
                   getAnimal());
31       }
32 }
```

Listing 4.6 – Exemple de cas de test JUnit seul pour l'aspect `AnimalComplaint-StateAspect`.

```
1  public class AdviceTracerTest extends TestCase {
2        public void testAnimalComplaintStateAspect3() {
3              AnimalComplaint complaint = new AnimalComplaint();
4              addTracedAdvice("AnimalComplaintStateAspect3");
5              setAdviceTracerOn();
6              complaint.setStatus(closed);
7              setAdviceTracerOff();
8              (*\textit{assertAdviceExecutionsEquals}*)("
                   AnimalComplaintStateAspect3",1);
9              (*\textit{assertExecutedAdviceAtJoinpoint}*)(
10                   "AnimalComplaintStateAspect3",
11                   "healthwatcher.model.complaint.
                         AnimalComplaint.setStatus:54");
12       }
13 }
```

Listing 4.7 – Exemple de cas de test AdviceTracer pour l'aspect `AnimalComplaint-StateAspect`.

```
1  @AdviceName("AnimalComplaintStateAspect3")
2  after(int status, AnimalComplaint animalComplaint,
3        AnimalComplaintState state):
4        execution(void AnimalComplaintState+.setStatus(int,
5                                        AnimalComplaint)) &&
6        args(status, animalComplaint) && target(state) {
7     if(status == open){
8        animalComplaint.setComplaintState(
9           new AnimalComplaintStateOpen(state)
10       );
11    } else if(status == closed){
12       animalComplaint.setComplaintState(
13          new AnimalComplaintStateClosed(state)
14       );
15    }
16 }
```

Listing 4.8 – Le greffon `AnimalComplaintStateAspect3`.

correct des points de jonction, et que le greffon a été tissé au bon endroit. Le cas de test écrit seulement avec JUnit réussit si la valeur de l'attribut `animal` n'a pas changée, ce qui peut arriver aussi bien parce que le greffon a été correctement tissé que parce qu'il y a une faute dans le programme.

En résumé, AdviceTracer réduit l'effort requis pour tester les expressions de point de coupe. Les cas de test écrit avec AdviceTracer ont été écrits plus rapidement que les cas de test écrit seulement avec JUnit. Les cas de test écrit avec AdviceTracer sont aussi plus simple.

Question 2 Le score de mutation de T_{AT} est meilleur que celui de T_{JU} (Figure 4.7). T_{AT} a atteint un score de mutation de 100 %, ce qui veut dire qu'elle a été capable de tuer tous les mutants, et donc de détecter toutes les fautes insérées par les opérateurs. T_{JU} n'a atteint qu'un score de 81,59 %, 14 mutants n'ayant pas été tués. Cela veut dire que T_{JU} n'a pas été capable de détecter toutes les fautes. Dans la suite, nous discutons les raisons pour lesquelles ces fautes n'ont pas été détectées.

Quatre des mutants non tués par T_{JU} sont des mutants d'un aspect qui implémente le patron de conception état. Après l'initialisation d'un objet héritant de la classe `Complaint`, son état est initialisé par le greffon. Avec ces mutants, le greffon est exécuté plusieurs fois à l'initialisation, ce qui peut-être détecté par AdviceTracer. Pour T_{JU} ce n'est pas possible, car le résultat de l'initialisation est le même pour les mutants que pour le système original.

Deux des mutants non tués par T_{JU} sont des mutants d'un aspect qui implémente le patron de conception «commande». À l'initalisation du système, les commandes sont enregistrées dans une table. Un des mutants tisse le greffon à différents endroits du code, et la table est donc réinitialisée à plusieurs reprise. Mais comme cette table est privée et n'est pas accessible à l'aide de méthodes publiques, il n'est pas possible de tuer ce mutant sans AdviceTracer. L'autre mutant tisse le greffon qui exécute une commande juste après que cette commande soit supprimée, ce qui provoque une erreur qui n'est pas levée, mais simplement ignorée par le système. Comme la commande est correctement retirée, il n'y a aucun comportement observable, et JUnit seul ne peut tuer ce mutant.

Les huit derniers mutants sont liés à un aspect qui synchronise plusieurs méthodes afin d'empêcher des exécutions concurrentes. Même s'il est possible de tuer ces mutants en utilisant seulement JUnit, créer un cas de test déterministe pour un comportement concurrent est difficile. Avec AdviceTracer ces mutants sont tués simplement, car il n'y a pas besoin de vérifier le comportement du greffon dans l'oracle.

En résumé, AdviceTracer améliore la capacité des cas de test à détecter des fautes dans les expressions de point de coupe. Avec JUnit, certaines fautes dans les expressions de point de coupe ne peuvent pas être détectées car le comportement observable du greffon associé ne permet pas d'exhiber une faute. AdviceTracer de son côté ne repose pas sur le comportement du greffon et est capable de détecter tous les types de fautes introduits par les opérateurs d'AjMutator.

4.3.5 Menaces à la validité

Dans cette section nous discutons des menaces à la validité interne et à la validité externe.

Les menaces à la validité interne sont les menaces qui peuvent biaiser les observations effectuées durant cette étude. Nous avons utilisé l'analyse de mutation comme critère de test pour écrire les suites de tests. Il est possible qu'avec d'autres critères, les suites des test aient été différentes. Cependant, AjMutator introduit différents types de fautes, identifiés par Ferrari *et al.* [Ferrari 08]. Les suites de tests ont donc du détecter des erreurs de types différents, et ont donc des cas de test variés. Nous pensons donc qu'il est peu probable que d'autres critères aient entraîné des résultats significativement différents.

Une autre menace à la validité interne est le fait que nous avons modifié le système HealthWatcher afin d'améliorer sa testabilité. Cependant, cette modification favorise la suite de tests écrite avec JUnit seulement. Puisqu'AdviceTracer ne requiert pas de vérifier le comportement des greffons, il est moins sensible aux problèmes de testabilité.

Les menaces à la validité externe sont les menaces qui peuvent biaiser la généralisation des résultats obtenus. HealthWatcher n'inclut pas tous les types d'expressions de point de coupe primitives, comme montré par la Table 4.1. Les études du Chapitre 2 montrent que l'utilisation des expressions de point de coupe primitives dans HealthWatcher est conforme à ce qu'on peut trouver dans la majorité des projets AspectJ. Nos résultats peuvent donc être généralisés à la majorité des projets AspectJ.

4.4 Conclusion

Les expressions de point de coupe d'AspectJ offre une grande flexibilité pour introduire des comportements transverses dans un programme Java. Cependant, elles sont aussi une source majeure d'erreurs ; une erreur dans une expression de point de coupe peut introduire des erreurs dans plusieurs endroits non-attendus ou manquer certains endroits attendus. Les expressions de point de coupe doivent être minutieusement vérifiées pour assurer la validité des programmes orientés-aspects. Avec les techniques existantes, JUnit est la seule possibilité pour vérifier automatiquement la validité des expressions de point de coupe. Nous avons montré que cela n'est pas suffisant, principalement car cela ne permet pas de cibler précisément les fautes dans les expressions de point de coupe. En conséquence, un cas de test JUnit censé capturer les fautes dans une expression de point de coupe peut manquer des fautes. Il peut aussi capturer des fautes qui ne se trouvent pas dans une expression de point de coupe, ce qui pose problème pour le diagnostique.

Nous avons proposé un outil, AdviceTracer, qui implémente une approche pour tester les expressions de point de coupe. AdviceTracer surveille les exécutions de greffons dans les programmes AspectJ et offre la possibilité de récupérer les informations à propos de ces exécutions à l'aide d'un petit ensemble d'assertions. Ces assertions peuvent être utilisées pour écrire des cas de test qui ciblent spécifiquement les fautes dans les

expressions de point de coupe. Afin d'évaluer cette approche, nous avons mené une étude empirique qui compare AdviceTracer avec JUnit pour l'écriture de cas de test pour des expressions de point de coupe AspectJ. Nous avons comparé deux dimensions des suites de tests : la complexité des cas de test, et leur capacité à détecter des fautes. L'étude empirique a été effectuée sur le système HealthWatcher. Nous avons observé que les cas de test qui utilisent AdviceTracer sont plus petits, requièrent moins de temps à écrire, ont des oracles qui ciblent mieux les fautes dans les expressions de point de coupe, et qu'ils détectent plus de fautes que les cas de test écrits avec JUnit seulement. Les oracles écrits avec AdviceTracer ne capturent que les fautes dans les expressions de point de coupe, et pas celles dans les greffons. Ceci permet d'améliorer l'interprétation des résultats des cas de test.

Dans de futurs travaux, nous aimerions explorer le cas de l'évolution des cas de test. Les expressions de point de coupe sont particulièrement vulnérables aux évolutions à cause de la fragilité de l'expression de point de coupe [Kellens 06]. Si de nouveaux points de jonction sont ajoutés lors de l'évolution du programme de base, des points de jonction peuvent être incorrectement sélectionnés par les expressions de point de coupe existantes, ou au contraire, des points de jonction peuvent être incorrectement ignorés par les expressions de point de coupe existantes. Il serait donc utile de mener une étude empirique pour observer comment des suites de tests écrites avec différentes techniques peuvent détecter de telless erreurs et évoluer avec le programme.

Chapitre 5

Formalisme et analyse de programmes orientés aspect

AspectJ est un langage orienté aspect basé sur Java, ainsi de nombreux outils d'analyse pour Java fonctionnent pour AspectJ. Par exemple, il est possible d'utiliser une syntaxe où les concepts ajoutés par AspectJ sont dans des annotations Java, pour utiliser des outils existants. Certains outils fonctionnent sur du bytecode et il est donc possible de les utiliser sur les fichiers produits par le compilateur AspectJ.

Même si l'utilisation d'outils pour Java peut être très utile, elle n'est pas toujours satisfaisante. En effet, ces outils ne peuvent pas raisonner correctement sur le programme car ils ne prennent pas en compte les concepts de la programmation orientée aspect. Par exemple, un outil mesurant des métriques de couplage sur des programmes Java ne prendra pas en compte le couplage dû au tissage d'aspects.

Dans le cadre de cette thèse, nous avons développé deux outils d'analyse pour des programmes AspectJ. Le premier, AjMetrics, est un outil de mesure de métriques pour programmes AspectJ. Le second, AjMutator, est un outil d'analyse de mutation pour les expressions de point de coupe AspectJ.

Les métriques sont des valeurs quantitatives qui permettent d'évaluer des propriétés qualitatives sur des programmes, telles que la testabilité, la modularité, la maintenabilité, etc. Les outils mesurant des métriques sur des programmes Java prennent en compte les concepts de la programmation orientée objet – tels que l'héritage, le polymorphisme etc. – mais pas les concepts de la programmation orientée aspect – tels que le tissage d'aspects ou les définitions inter-types –. Afin d'évaluer et de comprendre les propriétés de la programmation orientée aspect, il est nécessaire d'avoir des outils permettant de mesurer des métriques sur des programmes AspectJ, en tenant en compte des caractéristiques du langage.

Nous avons étendu un formalisme pour les programmes orientés aspect, qui permet de définir des métriques pour différents langages orientés aspect. Nous avons développé un outil de mesure de métriques sur les programmes AspectJ basé sur ce formalisme, AjMutator. Cet outil permet de mesurer différentes métriques sur des programmes AspectJ. Le formalisme que nous proposons est une contribution importante,

qui permet de définir un ensemble de métriques pour toute une famille de programmes orientés aspect.

L'analyse de mutation introduit des fautes dans un programme, afin d'évaluer une suite de tests. Les outils pour Java n'introduisent que des fautes identifiées pour Java. Or de nombreux types de fautes ont été identifiés pour AspectJ, et les langages orientés aspect en général, par Alexander *et al.* [Alexander 04] ou Ferrari *et al.* [Ferrari 08]. Afin de développer et d'évaluer des techniques de test spécifiques à la programmation orientée aspect, il est nécessaire d'avoir des outils d'analyse de mutation capable d'insérer des fautes spécifiques à la programmation orientée aspect.

Nous avons développé un outil d'analyse de mutation pour les expressions de point de coupe d'AspectJ, AjMutator. Cet outil produit des mutants en insérant des fautes dans les expressions de point de coupe, puis il les compile et exécute une suite de tests sur ces mutants. Lors de la compilation les mutants sont classés selon le type des fautes introduites. Une importante contribution de cet outil est que la classification permet, dans la majorité des cas, de détecter automatiquement les mutants équivalents. Dans le cas général, il s'agit d'un problème indécidable.

Ces deux outils ont besoin d'analyser les programmes Aspect en général, et les expressions de point de coupe en particulier. Afin de réduire le coût de construction de ces outils, nous avons développé deux composants d'analyse des programmes AspectJ. Ces deux composants correspondent à deux façons d'observer une expression de point de coupe. Le premier composant s'intéresse à la syntaxe d'une expression de point de coupe, qui permet de déclarer un ensemble de points de jonction. Le second composant s'intéresse aux points de jonction effectivement sélectionnés par les expressions de point de coupe, et abstrait le programme AspectJ dans son ensemble.

Ces deux composants d'analyses sont nécessaires à la fois pour mesurer les métriques et pour l'analyse de mutation. Dans le cas de la mesure des métriques, il est nécessaire de pouvoir mesurer l'utilisation des expressions de point de coupe, et donc d'analyser leur syntaxe, et d'analyser les programmes AspectJ pour les métriques de couplage. Dans le cas de l'analyse de mutation, les erreurs sont introduites dans les expressions de point de coupe, il faut donc un outil pour observer et modifier leur syntaxe. La classification des mutants s'effectue à partir de l'ensemble des points de jonction sélectionnés, il est donc nécessaire d'avoir une abstraction des programmes AspectJ.

Dans ce chapitre, nous détaillons les outils développés ainsi que les composants d'analyse utilisés. Dans la Section 5.1 nous détaillons le composant d'analyse syntaxique des expressions de point de coupe, puis dans la Section 5.2 nous détaillons le composant de modélisation qui permet d'abstraire l'expression de point de coupe comme l'ensemble des points de jonction sélectionnés. La Section 5.3 présente un outil permettant de mesurer différentes métriques sur les expressions de point de coupe. La Section 5.4 présente AjMutator, un outil d'analyse de mutation qui utilise les outils présentés précédemment. Enfin, la Section 5.5 conclut ce chapitre.

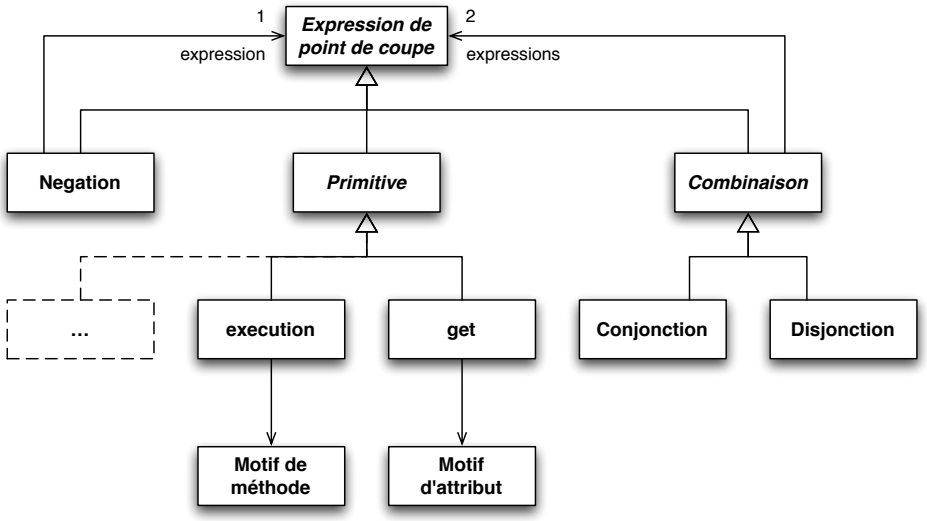

FIG. 5.1 – Un extrait du diagramme de classes de l'arbre syntaxique abstrait d'une expression de point de coupe.

5.1 Analyse syntaxique des expressions de point de coupe

L'objectif de l'analyse syntaxique d'une expression de point de coupe est d'obtenir, à partir d'une représentation textuelle, un modèle de la syntaxe de l'expression de point de coupe. La représentation que nous avons choisie est l'arbre syntaxique abstrait.

La Figure 5.1 montre un extrait du diagramme de classes de l'arbre syntaxique abstrait d'une expression de point de coupe. Une expression de point de coupe est soit une expression de point de coupe *primitive*, soit une combinaison de deux expressions de point de coupe, soit la négation d'une expression de point de coupe. Certaines expressions de point de coupe primitives ont besoin d'un ou plusieurs *motifs* comme paramètre. Par exemple une expression de point de coupe primitive `execution` a besoin d'un motif de méthode. La Figure 5.2 montre l'arbre syntaxique abstrait de l'expression de point de coupe `getAmount`.

À notre connaissance il n'existe pas d'analyseur syntaxique permettant de récupérer l'arbre syntaxique abstrait d'une expression de point de coupe afin de pouvoir l'observer ou la manipuler. Le compilateur AspectJ, ajc, utilise un analyseur *ad hoc*, écrit à la main, et n'offre pas d'interface permettant d'obtenir un arbre syntaxique abstrait. Nous avons donc développé un analyseur syntaxique pour les expressions de point de coupe AspectJ.

L'analyseur syntaxique a été développé en utilisant SableCC [Cagnon 98], un générateur d'analyseur syntaxique libre. SableCC permet de générer un arbre syntaxique abstrait automatiquement. Une fois la grammaire définie, dans un fichier texte, SableCC génère automatiquement un analyseur syntaxique ainsi que les classes de l'arbre syntaxique abstrait. L'analyseur syntaxique produit directement un arbre syntaxique abs-

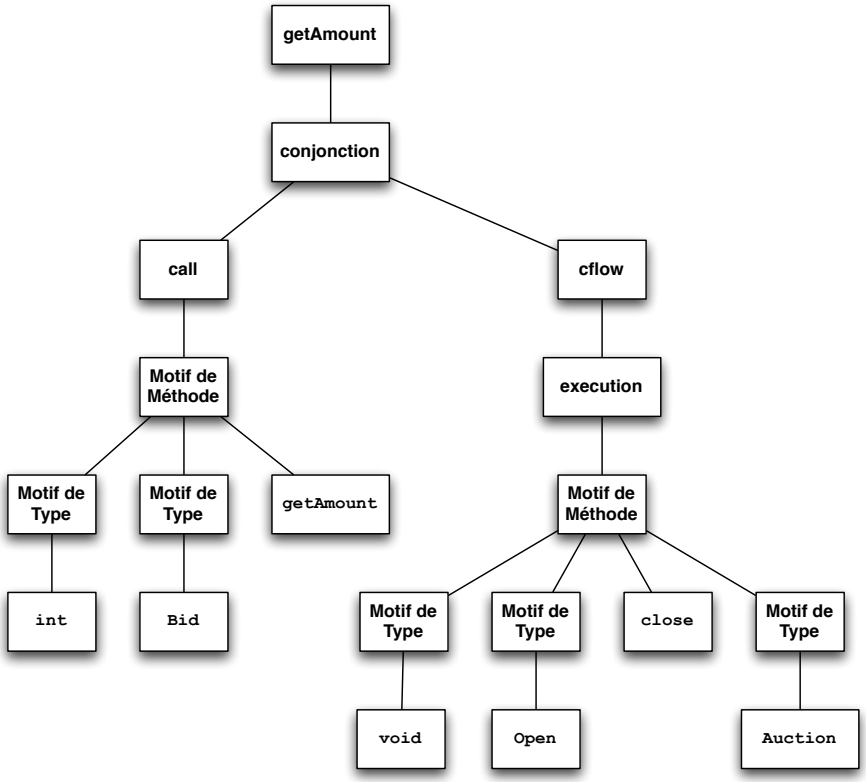

FIG. 5.2 – L'arbre syntaxique abstrait de l'expression de point de coupe `getAmount`.

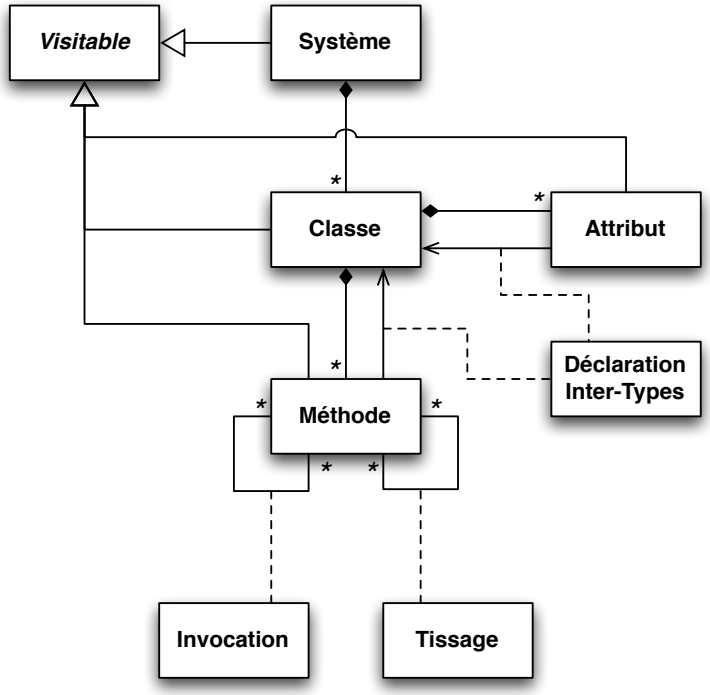

FIG. 5.3 – Méta-modèle de systèmes AspectJ

trait. Le patron de conception «visiteur» est implémenté sur l'arbre syntaxique abstrait, ce qui permet de le parcourir de manière générique, en minimisant le nombre de lignes de code.

Afin de pouvoir retrouver la forme textuelle d'un arbre syntaxique abstrait, nous avons développé un formateur. Il est parfois utile d'obtenir la représentation textuelle d'un arbre syntaxique abstrait, par exemple s'il a été modifié. Pour cela, le formateur est implémenté comme un «visiteur» de l'arbre syntaxique abstrait, qui retourne une chaînes de caractères à la fin du parcours de l'arbre.

5.2 Modélisation de programmes AspectJ

Pour obtenir un modèle des points de jonction sélectionnés par une expression de point de coupe, il est nécessaire de modéliser le système dans sa globalité, car les points de jonctions sélectionnés peuvent se trouver dans tout le programme.

Nous avons donc développé un composant permettant d'obtenir le modèle d'un système AspectJ, à partir de son code source. La Figure 5.3 montre le méta-modèle de systèmes AspectJ. Les aspects ne sont pas distingués des classes. En effet, les aspects d'AspectJ sont des classes spécialisées où les nouveaux concepts de la programmation orientée aspect sont définis. Dans le cas de notre abstraction, il n'est pas nécessaire de faire la distinction entre une classe et un aspect. De même, il n'est pas nécessaire de faire

la différence entre une méthode et un greffon. Comme les instances du méta-modèles sont construites à partir d'un programme AspectJ supposé correct, les modèles sont conformes par construction.

Un système est composé de classes, possédant des attributs et des méthodes. Les méthodes sont liées entre elles via deux relations. La première relation, avec la classe `Invocation`, correspond à l'invocation d'une méthode par une autre, tandis que la seconde, avec la classe `Tissage`, correspond au tissage d'un greffon dans une méthode. Le patron de conception «visiteur» est utilisé, afin de permettre un parcours simple d'un modèle.

Cette abstraction du programme permet d'observer où chaque greffon est tissé dans le programme de base. Les définitions *inter-types* sont aussi modélisées à l'aide d'une relation d'un attribut vers une classe, ou d'une méthode vers une classe.

Pour construire les modèles automatiquement à partir d'un système AspectJ, nous utilisons le compilateur AspectJ. Lors de la compilation, les expressions de point de coupe sont résolues, afin de tisser les greffons. Le compilateur fournit ensuite une interface qui permet de récupérer des informations sur le système, dont les points de jonction où les greffons sont tissés. Ces informations sont utilisées pour construire le modèle du système.

L'implémentation utilise EMF[1] (Eclipse Modeling Framework). Le méta-modèle a été produit à l'aide d'EMF et les différentes classes de la Figure 5.3 ont été générées automatiquement. Le composant est implémenté comme une extension pour l'IDE Eclipse. Lors de la compilation, l'extension récupère les informations nécessaires à partir du compilateur. Un modèle EMF conforme au méta-modèle est ensuite enregistré dans un fichier au format xmi.

5.3 Mesure de métriques

Nous avons utilisé un formalisme des programmes orientés aspect, afin de développer des outils adaptés à n'importe quel langage orienté aspect. Ce formalisme nous permet de définir des métriques métriques qui ne dépendent pas directement d'AspectJ. Les métriques sont définies et implémentées sur un modèle abstrait de programmes orientés aspect, puis un lien est définit entre le méta-modèle de programmes AspectJ, pour mesurer les métriques sur les modèles de programmes AspectJ.

Nous avons développé un outil basé sur ce formalisme, AjMetrics, pour mesurer des métriques sur les programmes AspectJ. Ces métriques sont définies aussi bien sur le système que sur les expressions de point de coupe en particulier. AjMetrics utilise donc aussi bien l'analyseur syntaxique d'expressions de point de coupe que les modèles de systèmes AspectJ.

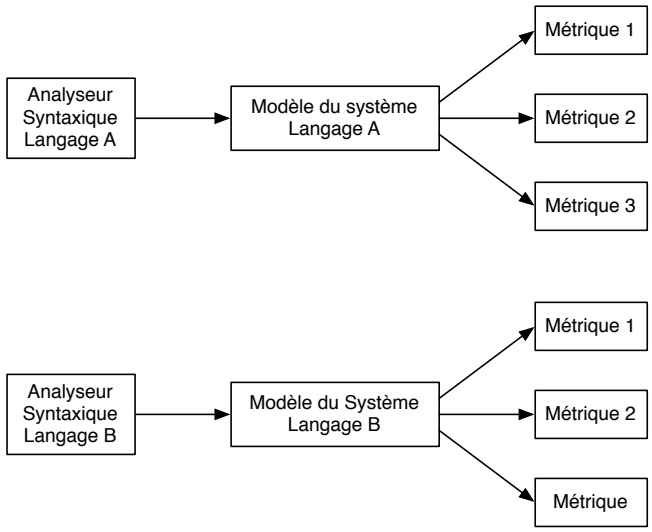

FIG. 5.4 – Construction d'un outil de mesure, sans formalisme.

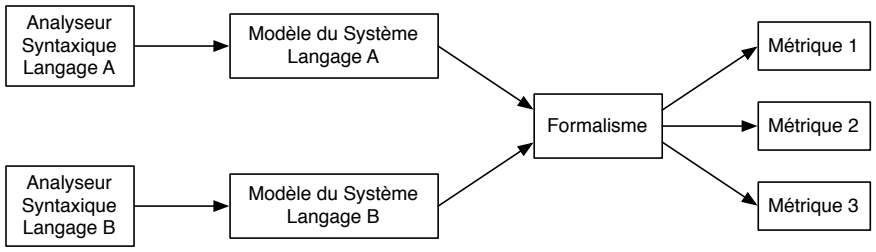

FIG. 5.5 – Construction d'un outil de mesure, avec un formalisme.

5.3.1 Formalisme des programmes orientés aspect

Un formalisme facilite la définition des métriques et la construction d'outils pour calculer ces métriques automatiquement. La Figure 5.4 illustre la construction d'un outil de mesure sans formalisme. Un analyseur syntaxique construit un modèle pour un système, et les métriques sont définies sur ce modèle. Le premier problème avec cette approche est qu'il peut être difficile de définir les métriques directement sur le modèle du système, selon la complexité du langage et du modèle. Le second problème est rencontré lorsqu'on veut que l'outil supporte un nouveau langage. Il faut construire un nouvel analyseur syntaxique, et redéfinir les métriques pour le nouveau modèle. S'il y a beaucoup de métriques à définir, cela peut devenir très coûteux. La Figure 5.5 illustre la construction d'un outil de mesure avec un formalisme. Il y a toujours besoin d'un nouvel analyseur syntaxique qui construit un modèle pour le nouveau langage, mais le modèle est traduit dans le formalisme, et les métriques sont définies sur le formalisme. Cela facilite la définition des métriques car le formalisme est de plus haut niveau que le modèle. Si on veut que l'outil supporte un nouveau langage, il faut toujours construire un nouvel analyseur syntaxique, mais au lieu de redéfinir toutes les métriques, il suffit de traduire le modèle du nouveau langage dans le formalisme. Les métriques sont ensuite automatiquement définies pour ce nouveau langage.

Briand *et al.* [Briand 99] ont proposé un formalisme pour définir des métriques pour des programmes orientés objet. Nous proposons d'étendre ce formalisme pour prendre en compte les aspects. Le but de cette extension est d'abstraire les systèmes écrits dans un langage à la fois orienté objet et orienté aspect.

Système

Définition 1 Système, classes, héritage. *Un système orienté aspect consiste en un ensemble de classes, C. Il peut exister des relations d'héritages entre les classes telles que pour chaque classe $c \in C$:*

- $Parents(c) \subset C$ *est l'ensemble des classes parentes de* c*.*
- $Children(c) \subset C$ *est l'ensemble des classes enfantes* c*.*
- $Ancestors(c) \subset C$ *est l'ensemble des classes ancêtres de* c*.*
- $Descendents(c) \subset C$ *est l'ensemble des classes descendantes de* c*.*

Méthodes

Définition 2 Méthodes d'une classe. *Pour chaque classe $c \in C$, soit $M(c)$ l'ensemble des méthodes de c.*

Définition 3 Méthodes déclarées et implémentées. *Pour chaque classe $c \in C$, soit :*

- $M_D(c) \subseteq M(c)$ *l'ensemble des méthodes déclarées dans* c*, c'est-à-dire les méthodes que* c *hérite mais ne redéfinit pas ou les méthodes virtuelles de* c*.*

[1] `http://www.eclipse.org/modeling/emf/`

- $M_I(c) \subseteq M(c)$ *l'ensemble des méthodes implémentées dans* c*, c'est-à-dire les méthodes que* c *hérite et redéfinit ou les méthodes non-virtuelles et non-héritées de* c*.*
- $M_A(c) \subseteq M(c)$ *l'ensemble des méthodes inter-types pour* c*, c'est-à-dire les méthodes de* c *déclarées et implémentées par une autre classe.*

où $\{M_D(c), M_I(c), M_A(c)\}$ *est une partition de* $M(c)$*.*

Définition 4 $M(C)$ *est l'ensemble des toutes les méthodes dans le système :*

$$M(C) = \bigcup_{c \in C} M(c)$$

Invocations

Définition 5 *Soit* $c \in C$*,* $m \in M_I(c)$*, et* $m' \in M(C)$*. Alors* $m' \in SIM(m) \Leftrightarrow \exists d \in C$ *telle que* $m' \in M(d)$ *et le corps de* m *invoque la méthode* m' *sur un objet de type statique* d*.*

Définition 6 *Soit* $c \in C$*,* $m \in M_I(c)$*, et* $m' \in SIM(m)$*.* $NSI(m, m')$ *est le nombre d'invocations par* m *de* m' *sur un objet de type statique* d*, et* $m' \in M(d)$*.*

Définition 7 *Soit* $c \in C$*,* $m \in M_I(c)$*, et* $m' \in M(C)$*. Alors* $m' \in PIM(m) \Leftrightarrow \exists d \in C$ *telle que* $m' \in M(d)$ *et le corps de* m *peut invoquer* m'*, à cause du polymorphisme et des liens dynamiques, sur un objet de type dynamique* d*.*

Définition 8 *Soit* $c \in C$*,* $m \in M_I(c)$*, et* $m' \in PIM(m)$*.* $NPI(m, m')$ *est le nombre d'invocations dans* m *où* m' *peut être invoquée sur un objet de type dynamique* d*, et* $m' \in M(d)$*.*

Greffons

Définition 9 Greffons tissés (WA). *Soit* $c \in C$*,* $m \in M_I(c)$*, et* $m' \in M(C)$*. Alors* $m' \in WA(m) \Leftrightarrow \exists d \in C, m' \in M_I(d)$ *et* m' *est tissé dans le corps de* m*.*

Définition 10 *Soit* $c \in C$*,* $m \in M_I(c)$*, et* $m' \in WA(m)$*.* $NWA(m, m')$ *est le nombre de fois où* m' *est tissé dans* m*.*

Les greffons sont des méthodes tissées dans d'autres méthodes.

Attributs

Définition 11 Attributs d'une classe. *Soit* $A(c)$ *l'ensemble des attributs de la classe* $c \in C$*.*

Définition 12 Attributs déclarés et implémentés. *Pour chaque classe* $c \in C$*, soit :*

- $A_D(c) \subseteq A(c)$ *l'ensemble des attributs déclarés dans la classe* c *(c'est-à-dire les attributs hérités).*
- $A_I(c) \subseteq A(c)$ *l'ensemble des attributs implémentés dans la classe* c *(c'est-à-dire les attributs non-hérités).*

– $A_A(c) \subseteq A(c)$ *l'ensemble des attributs inter-types de* c*, c'est-à-dire les attributs de* c *définits par une autre classe.*

où $\{A_D(c), A_I(c), A_A(c)\}$ *est une partition de* $A(c)$*.*

Définition 13 $A(C)$ *est l'ensemble de tous les attributs du système :*

$$A(C) = \bigcup_{c \in C} A(c)$$

Définition 14 *Soit* $AR(m)$ *l'ensemble des attributs référencés par la méthode* $m \in M(C)$*.*

Prédicat

Définition 15 Utilise. *Soit* $c \in C$*,* $d \in C$*.*

$$\begin{aligned} utilise(c,d) \quad \Leftrightarrow \quad & (\exists m \in M_I(c), \exists m' \in M_I(d), m' \in PIM(m)) \vee \\ & (\exists m \in M_I(c), \exists a \in A_I(d), a \in AR(m)) \vee \\ & (\exists m \in M_I(c), \exists m' \in M_I(d), m' \in WA(m)) \end{aligned}$$

Ce prédicat indique qu'une classe c *utilise* une classe d, si elle invoque une méthode définie par d, si elle référence un attribut de d, ou si d a une méthode qui est tissée dans c.

Discussion

Il existe plusieurs langages orientés aspect qui ont tous des spécificités et des concepts communs. La plupart de ces langages peuvent être abstraits à l'aide du formalisme proposé.

Classes spécifiques Certains langages introduisent un nouveau type de classe, utilisé pour encapsuler les nouveaux concepts de la programmation orientée aspect. En AspectJ ces classes spécifiques sont appelées *aspects* (mot-clé «`aspect`»). En CaesarJ ces classes sont appelées *cclass* (mot-clé «`cclass`»). D'autres langages comme AspectWerkz ou JBoss AOP n'ont pas de classe spécifique.

Dans le formalisme proposé, les classes et les classes spécifiques ne sont pas distinguées. Les différences sont très spécifiques et parfois seulement syntaxiques, il n'y a donc pas d'intérêt à formaliser ces différences.

Greffons Dans la plupart des langages orientés objet, un greffon est une méthode et ce qui est tissé est une invocation de cette méthode. En AspectJ et CaesarJ, une syntaxe spécifique est utilisé, tandis qu'en JBoss AOP et AspectWerkz les greffons sont des méthodes java.

Dans le formalisme proposé, les greffons et les méthodes ne sont pas distinguées, pour les mêmes raisons que pour les classes spécifiques.

Expression de point de coupe Les expressions de point de coupe sont le concept de la programmation orientée aspect qui diffère le plus entre les différents langages orientés aspect. Ils peuvent être définis en utilisant une syntaxe spéciale (AspectJ, CaesarJ), des annotations (AspectJ, AspectWerkz, JBoss AOP), ou xml (AspectWerkz, JBoss AOP).

Dans ce formalisme, nous considérons que les expressions de point de coupe ont été résolues et il y a une relation entre les méthodes pour déterminer les méthodes qui sont tissées dans d'autres méthodes. Le concept d'expression de point de coupe n'est donc pas directement formalisé.

Résumé des extensions

Le formalisme proposé est une extension du formalisme proposé par Briand *et al.* [Briand 99] pour prendre en compte les aspects. Si le système n'a pas d'aspect, les deux formalismes sont équivalents. Voici un résumé des changements apportés.

Définitions inter-types Les définitions inter-types sont des méthodes ou des attributs d'une classe qui sont définis par un aspect. L'extension pour les définitions inter-types consiste en deux ensembles, $M_A(c)$ et $A_A(c)$. $M_A(c)$ est l'ensemble des méthodes de c définies à l'extérieur de c par des aspects. $A_A(c)$ est l'ensemble des attributs de c définis à l'extérieur de c par des aspects.

Greffons Comme expliqué dans la discussion de la section précédente, les greffons sont des méthodes. L'extension pour les greffons consiste en $WA(m)$ et $NWA(m, m')$. $WA(m)$ est l'ensemble des greffons tissés dans la méthode m. $NWA(m, m')$ est le nombre de fois où m' est tissé dans m. Le prédicat $utilise$ a aussi été modifié pour prendre en compte les greffons.

5.3.2 Métriques définies sur le système AspectJ

Les métriques sur le système AspectJ, dont une définition informelle a été donnée en Section 2.1.3, sont définies formellement en utilisant le formalisme décrit précédemment.

Définitions

Couplage entre les objets (*CBO*) :

$$CBO(c) = |\{d \in C - \{c\} | uses(c, d) \vee uses(d, c)\}|$$

CBO a été défini par Chidamber et Kemerer [Chidamber 94]. Pour une classe *c*, *CBO* est le nombre de classes qui utilisent *c* ou qui sont utilisées par *c*. Cette définition est celle de Briand *et al.* [Briand 99]. Les aspects sont pris en compte, car le prédicat *uses* prend en compte les greffons.

Réponse à la classe (*RFC, RFC'*) :

$$\begin{aligned} R_0(c) &= M(c) \\ R_{i+1}(c) &= \bigcup_{m \in R_i(c)} PIM(m) \cup WA(m) \\ RFC_\alpha(c) &= \left| \bigcup_{i=0}^{\alpha} R_i(c) \right| \\ RFC'(c) &= RFC_\infty(c) \\ RFC(c) &= RFC_1(c) \end{aligned}$$

La métrique *RFC* a été définie par Chidamber et Kemerer [Chidamber 94], et est le nombre de méthodes d'une classe plus le nombre de méthodes pouvant être invoquées par cette classe. La métrique *RFC'* est la clôture transitive de *RFC*.

Les méthodes inter-types sont prises en compte car elles sont inclues dans $M(c)$. Les greffons tissés sont aussi pris en compte car on considère qu'ils sont invoqués par les méthodes où ils sont tissés.

Couplage par envoi de messages (*MPC*) :

$$MPC(c) = \sum_{m \in M_I(c)} \sum_{m' \in SIM(m) - M_I(c)} NSI(m, m') + NWA(m, m')$$

MPC a été définie par Li et Henry [Li 93] comme le nombre d'envois de messages dans une classe. Comme pour *RFC*, on considère que les greffons sont invoqués par les méthodes où ils sont tissés.

Manque de cohésion des méthodes (**LCOM**) :

$$LCOM(c) = \frac{\left(\sum_{a \in A_I(c)} \frac{|\{m \in M_I(c) | a \in AR(m)\}|}{|A_I(c)|} \right) - |M_I(c)|}{1 - |M_I(c)|}$$

La métrique *LCOM* est basé sur la définition de Henderson-Sellers [Henderson-Sellers 96] et a été définie dans le formalisme de Briand *et al.* par Bruntink et van Deursen [Bruntink 04]. La valeur de *LCOM* est zéro si tous les attributs de c sont référencés par toutes les méthodes de c. Ceci indique une cohésion parfaite. La valeur de *LCOM* est un s'il y a un manque complet de cohésion. Dans ce cas, chaque attribut n'est référencé que par une méthode. *LCOM* ne peut pas être calculée s'il n'y a pas d'attribut ou s'il n'y a qu'une seule méthode dans la classe, car cela provoquerait une division par zéro.

Implémentation

L'implémentation du formalisme de programmes orientés aspect ne s'appuie pas directement sur le méta-modèle des programmes AspectJ. L'intérêt de définir un for-

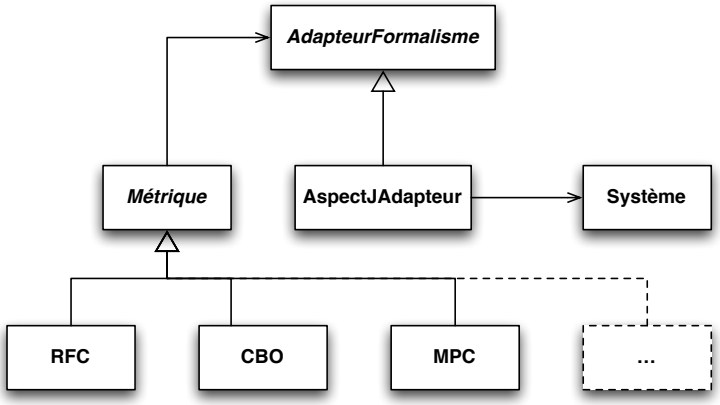

FIG. 5.6 – Le diagramme de classes des différentes métriques.

malisme est de pouvoir réutiliser les métriques sur d'autres langages. Nous avons donc utilisé le patron de conception *adapteur* [Gamma 95], comme illustré sur le diagramme de classes de la Figure 5.3.

Une classe abstraite *Métrique* est parente de toutes les métriques. Ces classes accèdent aux informations du système via une classe *AdapteurFormalisme*, qui fournit une interface permettant d'obtenir les différents ensembles et prédicats définis précédemment. Cette classe est abstraite, et il faut écrire un adapteur pour chaque langage. Pour AspectJ, nous avons écrit la classe *AspectJAdapteur* qui fait le lien entre le formalisme et le méta-modèle de programmes AspectJ (la classe *Système* est celle du méta-modèle de la Figure 5.3).

Ainsi, pour mesurer des métriques sur un nouveau langage, il faut un outil qui puisse produire un modèle d'un programme écrit avec ce nouveau langage, et un nouvel adapteur entre les métriques et le modèle. Les classes des métriques (et donc leurs définitions) restent identiques.

5.3.3 Métriques définies sur la syntaxe

Ces métriques sont mesurées sur la syntaxe des expressions de point de coupe, et sont définies sur l'arbre syntaxique abstrait des expressions de point de coupe d'AspectJ. Ces métriques sont spécifiques à AspectJ, car la syntaxe des expressions de point de coupe d'AspectJ est spécifique.

Elles sont définies à l'aide du patron de conception «visiteur». Chaque métrique est définie dans une classe qui hérite du visiteur de l'arbre syntaxique abstrait.

Les métriques définies sur la syntaxe ont pour but de mesurer l'usage de chaque expression de point de coupe primitives, ainsi que des combinaisons des expressions de point de coupe. Ce sont les métriques *NPCD*, Section 2.1.3.

5.4 Analyse de mutation

À l'aide de l'analyseur syntaxique et des modèles de systèmes AspectJ présentés dans les sections précédentes, nous avons développé un outil pour l'analyse de mutation de programmes AspectJ.

L'analyse de mutation consiste à évaluer une suite de cas de test en insérant des fautes dans un programme. Les fautes sont insérées par des opérateurs de mutation, chacun insérant un type de faute différent et défini. Différents mutants sont créés, une faute étant insérée par mutant. Un mutant est considéré comme «tué» par une suite de cas de test si cette suite est capable d'exhiber une différence entre le programme original et le mutant. Le score de mutation d'une suite de cas de test est le pourcentage de mutants tués par cette suite.

L'analyse de mutation des expressions de point de coupe permet d'évaluer des suites de cas de test ciblant les expressions de point de coupe et donc d'évaluer les techniques pour tester les expressions de point de coupe.

À notre connaissance il n'existe aucun outil permettant d'effectuer une analyse de mutation d'expressions de point de coupe. Un tel outil doit pouvoir créer des mutants en insérant des fautes à l'aide d'opérateurs de mutation, de compiler ces mutants et d'exécuter des cas de test sur ces mutants, afin d'obtenir un score de mutation.

Nous avons donc développé un outil pour l'analyse de mutation d'expressions de point de coupe, AjMutator. Il s'appuie sur des opérateurs de mutation définis par Ferrari *et al.* [Ferrari 08].

AjMutator classe automatiquement les mutants, en analysant les points de jonction sélectionnés par l'expression de point de coupe originale et l'expression de point de coupe mutante. Il est possible de classer les mutants selon le type des fautes introduites, de manière analogue aux types de fautes définis précédemment, voir la Section 4.1 et la Figure 4.1. Les mutants sont classés à la compilation, en analysant statiquement les expressions de point de coupe.

La classification des mutants permet, dans la majorité des cas, de détecter automatiquement les mutants équivalents. En effet, un mutant équivalent est un mutant qui sélectionne les mêmes points de jonction que l'expression de point de coupe original. Dans le cas d'expression de point de coupe statique, ces mutants sont détectables statiquement. Il s'agit d'une contribution importante, car le problème des mutants équivalents est indécidable dans le cas général.

5.4.1 Analyse de mutation des expressions de point de coupe AspectJ

Une expression de point de coupe mutante est une expression de point de coupe qui sélectionne un ensemble de points de jonction différent de l'ensemble de points de jonction sélectionné par l'expression de point de coupe originale. Si l'ensemble est identique alors il n'y a pas de faute, et s'il est différent cela peut provoquer des effets de bord indésirables.

Ferrari *et al.* [Ferrari 08] ont proposé différents opérateurs de mutation pour les pro-

TAB. 5.1 – Opérateurs implémentés dans AjMutator.

Operator	Description
PCCC	Remplace un `cflow` par un `cflowbelow`, et vice-versa
PCCE	Remplace un `call` par un `execution`, et vice-versa
PCGS	Remplace un `get` par un `set`, et vice-versa
PCLO	Modifie les opérateurs logiques dans une composition
PCTT	Remplace un `this` par un `target`, et vice-versa
POEC	Ajoute, retire, ou modifie les clauses d'exceptions
POPL	Modifie la liste des paramètres des expressions primitives
PSWR	Retire des *jokers*
PWAR	Retire les annotations des déclarations de type, champs, méthode, et constructeur
PWIW	Ajoute des *jokers*

grammes AspectJ. Ces opérateurs sont basés sur des types de fautes identifiés dans différents travaux. Les auteurs présentent trois types d'opérateur : les opérateurs pour les expressions de point de coupe, les opérateurs pour les déclarations AspectJ, et les opérateurs pour les définitions et implémentations des greffons. Pour AjMutator, nous ne considérons que les opérateurs pour les expressions de point de coupe. La Table 5.1 décrit les opérateurs implémentés dans AjMutator.

Classification des mutants

Les expressions de point de coupe mutantes sont classées en comparant l'ensemble des points de jonction qu'elles sélectionnent avec l'ensemble des points de jonction sélectionnés par l'expression de point de coupe originale. La classification des mutants est analogue à la classification des fautes de la Figure 4.1, page 76.

Un mutant qui sélectionne des points de jonction non-attendus et ne sélectionne pas certains points de jonction attendus, et introduit donc une faute de type 1, est un mutant de classe 1. Un mutant qui ne sélectionne pas certains points de jonction attendus, et introduit donc une faute de type 2, est un mutant de classe 2. Un mutant qui sélectionne des points de jonction non-attendus, et introduit donc une faute de type 3, est un mutant de classe 3. Enfin, un mutant qui sélectionne les même points de jonction que l'expression de point de coupe originale est un mutant équivalent, il n'introduit pas de faute.

Il est important de remarquer que la classification des mutants dépend du système dans lequel ils sont tissés. La même expression de point de coupe mutante peut avoir deux classifications différentes dans deux systèmes différents.

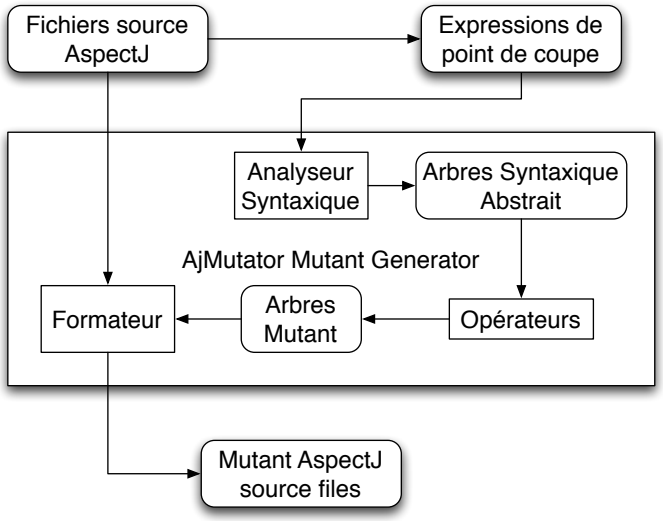

FIG. 5.7 – Processus de génération des mutants d'AjMutator

5.4.2 AjMutator

AjMutator est un outil pour l'analyse de mutation d'expressions de point de coupe AspectJ. Il produit des mutants en insérant des fautes dans les expressions de point de coupe.

AjMutator est séparé en trois parties distinctes :

1. la génération des fichiers sources mutants à partir de fichiers sources AspectJ
2. la compilation des fichiers sources mutants
3. l'exécution des cas de test sur les mutants pour calculer le score de mutation de la suite de cas de test

Génération des mutants

La Figure 5.7 présente le processus de génération des mutants d'AjMutator. L'analyseur syntaxique, décrit dans la Section 5.1, construit un arbre syntaxique abstrait pour chaque expression de point de coupe dans le code source AspectJ. Les opérateurs insèrent des fautes dans des copies de l'arbre syntaxique abstrait, il y a donc un arbre syntaxique abstrait original et un arbre syntaxique abstrait pour chaque mutant. Un formateur produit ensuite un fichier source AspectJ pour chaque mutant, en traduisant l'arbre syntaxique abstrait de chaque mutant en format texte et en l'insérant à la place de l'original. Au final, on obtient un ficher source pour chaque mutant.

Les opérateurs de mutation sont implémentés en utilisant le patron de conception «visiteur» [Gamma 95]. Chaque opérateur étend la classe abstraite `Operator`, qui est un visiteur pour l'arbre syntaxique abstrait des expressions de point de coupe. Lorsqu'un opérateur rencontre un endroit où il peut insérer une faute, un arbre syntaxique

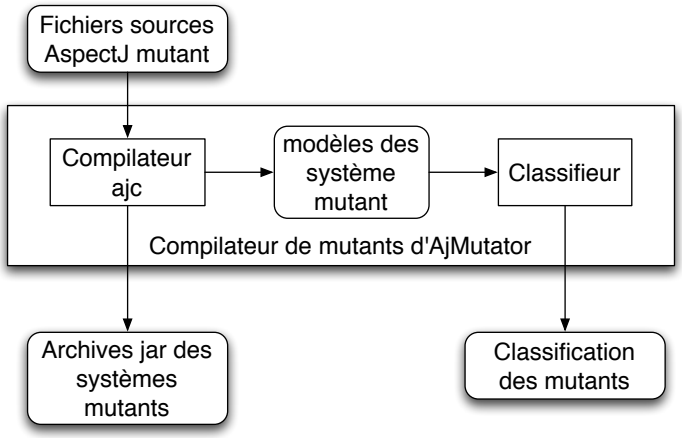

FIG. 5.8 – Processus de compilation et de classification des mutants d'AjMutator.

abstrait mutant est généré. L'arbre syntaxique abstrait mutant est ensuite formaté sous forme de texte et inséré dans une copie du fichier source original. Pour réduire la consommation mémoire, l'arbre syntaxique abstrait est immédiatement formaté et la référence est effacée pour que le ramasse-miette puisse libérer la mémoire.

Compilation des mutants

L'étape suivant la génération des mutants est leur compilation. La compilation est requise pour pouvoir exécuter les cas de test sur les systèmes mutants, mais aussi pour classer les mutants automatiquement.

La Figure 5.8 présente le processus de compilation et de classification des mutants d'AjMutator. Ce processus repose sur le compilateur d'AspectJ, ajc[2]. Le compilateur produit une archive `jar` du système pour chaque mutant.

Comme la modification d'une expression de point de coupe peut affecter différentes classes dans tout le système, il est nécessaire de garder une version compilée de tout le système pour chaque mutant.

Un modèle du système est crée pour chaque mutant. Ce modèle, tel que décrit dans la Section 5.2, permet de classer automatiquement les mutants, en analysant l'ensemble des points de jonction sélectionnés.

La Figure 5.9 présente le processus de classification des mutants. D'abord, si le mutant n'est pas compilable il n'est pas sélectionné car l'analyse de mutation ne concerne que les mutants exécutables. Si le mutant produit des points de jonction négligés et des points de jonction non-attendus, il est de classe 1. Si le mutant produit des points de jonction négligés mais pas de point de jonction non-attendu, il est de classe 2. Si le mutant produit des points de jonction non-attendus mais pas de point de jonction négligé, il est de classe 3. Enfin si le mutant ne produit ni de point de jonction négligé ni de

[2] `http://www.eclipse.org/aspectj`

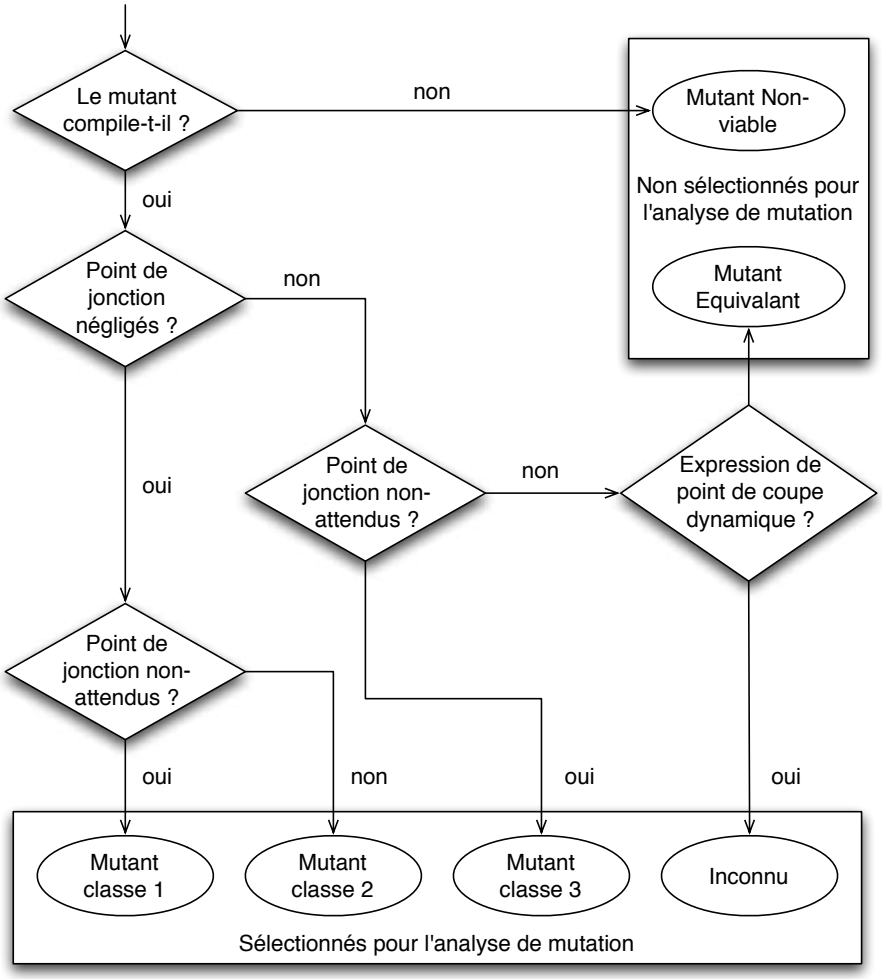

FIG. 5.9 – Processus de classification des mutants.

point de jonction non-attendu, il est équivalent et n'est pas sélectionné pour l'analyse de mutation.

La précision du processus de classification est différente selon que l'expression de point de coupe originale soit statique ou dynamique. Ceci est résumé de la façon suivante :

Expression de point de coupe statique L'ensemble des points de jonction sélectionnés est calculable statiquement, et la classification est donc exacte.

Expression de point de coupe dynamique L'ensemble des points de jonction sélectionnés ne peut être que sur-approximé, la classification n'est donc pas parfaitement précise. Deux cas peuvent être distingués :

Non-équivalent Si le mutant est classé comme non-équivalent, cela veut dire que la partie statique a été modifiée. Même s'il n'y a aucune certitude, il est probable que le mutant soit non-équivalent, et que la classification soit exacte.

Équivalent Si le mutant est classé comme équivalent cela veut juste dire que la partie statique du mutant est sémantiquement équivalente à la partie statique de l'original, mais les parties dynamiques peuvent être différentes. Dans ce cas il faut sélectionner le mutant pour l'analyse de mutation.

Comme montré sur la Figure 5.9, avant de classé un mutant comme équivalent, on vérifie d'abord si le mutant original est statique ou dynamique. S'il est dynamique, il n'est pas classé et le mutant est sélectionné pour l'analyse de mutation.

Exécution des cas de test

Le but d'une analyse de mutation est d'évaluer une suite de cas de test à l'aide d'un score de mutation. Le score de mutation est le ratio du nombre de mutants tués sur le nombre total de mutants. Le score de mutation évalue la capacité d'une suite de cas de test à détecter des fautes dans le système muté.

Un mutant est considéré comme tué si la suite de cas de test peu exhibé une différence entre le système original et le système mutant. Donc si le mutant est tué, la suite de cas de test peut détecter l'erreur introduite tandis que si le mutant est toujours vivant, la suite de cas de test n'est pas capable de détecter cette faute et il est nécessaire de l'améliorer, soit en ajoutant des données de test, soit en améliorant l'oracle, ou les deux.

AjMutator repose sur JUnit pour les cas de test et suppose que tous les cas de test réussissent sur le système original. Un mutant est considéré comme tué si au moins un cas de test échoue, car le résultat est alors différent de celui de l'original.

5.5 Conclusion

La validation expérimentale de nos propositions pour le test de programmes orientés aspect constitue une contribution importante de la thèse. Ces expériences nécessitent des outils qui ne sont pas toujours disponibles. Nous avons donc développé

plusieurs outils d'analyse de programmes orientés aspect au cours de cette thèse. Ce chapitre synthétise cette contribution technique à travers la présentation de différents outils pour l'analyse des expressions de point de coupe. AjMetrics, est un outil permettant de mesurer différentes métriques sur les programmes AspectJ. Ces métriques sont définies aussi bien sur le système global que sur les expressions de point de coupe en particulier. AjMutator est un outil pour l'analyse de mutation des expressions de point de coupe AspectJ. Il insère des fautes dans les expressions de point de coupe, puis compile et classe les mutants, pour finalement exécuter des suites de tests sur les mutants.

Ces deux outils ont besoin d'analyser les expressions de point de coupe, sous deux formes différentes. Les expressions de point de coupe peuvent être observées du point de vue de leur syntaxe, ou du point de vue des points de jonction qu'elles sélectionnent.

Des composants d'analyses, communs à AjMetrics et AjMutator ont donc été développé. Le premier composant analyse une expression de point de coupe afin de produire un arbre syntaxique abstrait décrivant la syntaxe. Le second composant analyse les informations fournies par le compilateur AspectJ afin de produire un modèle du système permettant d'observer où les greffons ont été tissés.

Conclusion et perspectives

Dans cette thèse, nous avons proposé des solutions pour le test de programmes orientés aspect. Ces solutions ont été élaborées à la suite d'observations effectuées lors d'études empiriques de programmes orientés aspect. Cette thèse comporte ainsi quatre contributions principales.

La première contribution est **l'étude empirique de programmes orientés aspect**. Cette étude, menée sur 38 projets libres écrits à l'aide d'AspectJ, nous a permis d'observer que les développeurs utilisent peu de greffons, et que ces greffons sont tissés à peu d'endroits. De plus, seule une partie du langage d'expression de point de coupe d'AspectJ est réellement utilisée. En comparant des implémentations Java et AspectJ d'un même programme, nous avons constaté que si la programmation orientée aspect permet bien d'améliorer la modularité, elle diminue la testabilité.

Ces observations ont montré que les développeurs manquent de confiance dans la programmation orientée aspect. Nous avons donc fait le choix de proposer des solutions pour aider les développeurs, en ciblant en particulier le test de programmes orientés aspect. En effet, le test d'un logiciel est une étape cruciale du développement permettant de s'assurer de sa qualité, or nous avons observé que les programmes orientés aspect sont moins testables.

La seconde contribution de cette thèse est **une analyse statique de l'impact des aspects sur les cas de test**. L'objectif est, dans le cadre d'un développement incrémental où le programme de base est testé avant les aspects, de pouvoir réutiliser le maximum de cas de test écrits à l'étape de test précédente, en étant sûr que ces cas de test n'ont pas besoin d'être modifiés. L'analyse statique est ainsi capable, avec des sur-approximations, de repérer les cas de test qui ont été impactés par l'introduction d'aspects. Ces cas de test doivent être modifiés pour prendre en compte les aspects, les autres peuvent être réutilisés tels quel.

Afin de mener des expériences, cette analyse statique a été implémentée dans un outil, Vidock. Nos études ont montré que Vidock est capable de détecter les cas de test impactés avec efficacité : tous les cas de test impactés sont détectés, et très peu de cas de test non-impactés sont détectés comme impactés. Ceci montre que les sur-approximations, liés à la liaison dynamique et aux expressions de point de coupe dynamiques, ont peu d'effets sur les résultats de l'analyse.

La troisième contribution de cette thèse est **une approche pour la définition d'oracles précis pour le test d'expressions de point de coupe**. Habituellement, lors du test

d'expressions de point de coupe, l'oracle doit vérifier la présence ou l'absence d'un greffon en vérifiant son comportement. Ceci peut créer des effets de bord ne permettant pas de localiser efficacement une faute lors de l'échec d'un cas de test. Nous avons donc développé une approche, implémentée dans un outil nommé AdviceTracer, permettant de définir des oracles précis pour le test d'expressions de point de coupe. Lors de l'exécution d'un cas de test, AdviceTracer enregistre les différentes exécutions des greffons, et fournit des assertions permettant de vérifier la présence ou l'absence d'un greffon à un point de jonction précis. Notre approche permet d'interpréter directement l'échec d'un cas de test.

Nous avons mené une étude empirique, visant à démontrer les intérêts de notre approche par rapport au test traditionnel d'expressions de point de coupe. Pour cela, deux développeurs ont testé le même système en utilisant chacun une technique de test différente. Le premier développeur n'a utilisé que JUnit, tandis que le second a utilisé AdviceTracer. Cette étude montre que les cas de test écrits à l'aide d'AdviceTracer sont plus précis, plus simples, et sont plus rapides à écrire. De plus, les cas de test AdviceTracer sont capables de détecter des erreurs que les cas de test JUnit ne peuvent pas détecter.

La quatrième contribution est **un formalisme pour la définition de métriques sur les programmes orientés aspect**. Ce formalisme se veut le plus générique possible afin de s'adapter à un maximum de langages orientés aspect différents. Ce formalisme, basé sur le travail de Briand *et al.* [Briand 99], a été implémenté dans un outil permettant de mesurer diverses métriques sur des programmes orientés aspect, AjMetrics. Cet outil a été utilisé lors d'études empiriques sur les programmes orientés aspect.

Nous avons aussi implémenté un outil pour l'analyse de mutation d'expressions de point de coupe AspectJ, AjMutator. En effet, lors de la validation de technique de test pour les expressions de point de coupe, nous avons eu besoin de qualifier des suites de test à l'aide d'une analyse de mutation. À notre connaissance, il n'existait aucun outil, nous avons donc développé un AjMutator, en utilisant des opérateurs de mutation présentés par Ferrari *et al.* [Ferrari 08]. AjMutator est capable, dans le cas d'expressions de point de coupe statiques, de détecter les mutants équivalents automatiquement.

Perspectives

Les perspectives pour les travaux futurs peuvent être dans la continuité directe des travaux que nous avons effectués, ou concerner le test de programmes orientés aspect de manière plus générale.

Il serait intéressant de pouvoir automatiquement inférer la façon dont les cas de test impactés doivent évoluer. Dans le Chapitre 3, nous avons présenté une analyse statique permettant de détecter automatiquement quels sont les cas de test impactés par l'introduction d'aspects. Ces cas de test impactés doivent ensuite être modifiés pour prendre en compte les aspects. Nous pensons que la façon dont les cas de test impactés doivent être modifiés dépend des aspects qui les impactent.

Nous proposons donc d'utiliser une classification des aspects, telle celle présentée

par Munoz *et al.* [Munoz 08]. Cette classification devrait permettre d'inférer le résultat attendu des cas de test qu'impacte un aspect. Par exemple, certains aspects ne font qu'augmenter le comportement du programme, sans affecter le comportement existant. On s'attend donc à ce que les cas de test impactés par un tel aspect réussissent toujours après le tissage. Il est par contre nécessaire de prendre en compte le nouveau comportement, par exemple en augmentant les oracles des cas de test existants.

Concernant le test des expressions de point de coupe, une perspective concerne l'évolution du programme et des aspects. Un problème de la programmation orientée aspect est la fragilité de l'expression de point de coupe qui peut ne plus être correcte après l'évolution du programme de base. Dans le Chapitre 4, nous présentons une approche, basée sur AdviceTracer, pour écrire des oracles précis pour les expressions de point de coupe.

Dans le futur, nous envisageons donc d'étudier comment se comporte les cas de test écrits à l'aide d'AdviceTracer vis à vis de l'évolution du programme. Dans le cas où l'expression de point de coupe évolue, les cas de test doivent être modifiés afin de prendre en compte les changements. Lors de l'étude empirique que nous avons menée, nous avons comparé les cas de test écrits avec JUnit et ceux écrits avec AdviceTracer, mais cette comparaison n'incluait pas l'évolutivité des cas de test. Il serait donc intéressant d'observer quelle suite de cas de test est la plus simple à faire évoluer avec l'expression de point de coupe.

Lorsque le programme de base évolue, il est possible qu'une expression de point de coupe ne soit plus correcte. Nous envisageons d'observer dans quelle mesure les cas de test écrits avec AdviceTracer peuvent détecter une telle erreur, et donc être une solution à la fragilité de l'expression de point de coupe. Si après l'évolution, l'expression de point de coupe sélectionne par erreur de nouveaux points de jonction, il faut que les cas de test écrits pour détecter les points de jonction non-attendus couvrent les nouveaux points de jonction. Si l'expression de point de coupe est censée capturer de nouveaux points de jonction, il faut écrire de nouveaux cas de test pour les points de jonction attendus.

Une partie du test de programmes orientés aspect n'a pas été traitée dans cette thèse, il s'agit du test des greffons, qui a déjà été considéré par de nombreux travaux [Xie 06b, Harman 09, Zhao 03, Xu 06, Lemos 07, Lopes 05]. Une perspective que nous envisageons est d'étudier les liens entre les cas de test pour les greffons et les cas de test pour les expressions de point de coupe.

Des critères minimum de test ont été proposés pour le test de greffons [Lemos 07]. Nous envisageons d'étudier comment ces critères minimum peuvent être utilisés pour écrire des cas de test visant spécifiquement les expressions de point de coupe. Si ces critères minimum ne sont pas suffisants, il est nécessaire d'en proposer de nouveaux, adaptés au test des expressions de point de coupe. Si ces critères sont suffisants, il est possible de réutiliser une partie des scénarios et données de test pour tester les greffons et les expressions de point de coupe.

Enfin il est important d'étudier comment le test de programmes orientés aspect doit être organisé. Nous avons proposé une approche incrémentale, où le programme de

base est d'abord testé seul, puis les aspects sont tissés et testés successivement. Lors du test des aspects plusieurs solutions restent possibles. Par exemple, il est possible de tester d'abord les expressions de point de coupe puis les greffons, ou au contraire tester d'abord les greffons. Il est aussi possible qu'une solution soit d'écrire les cas de test pour les greffons et pour les expressions de point de coupe en même temps, si par exemple ils partagent les données de test.

Glossaire

Aspect Un aspect est une unité qui encapsule une préoccupation transverse. Un aspect est composé d'une expression de point de coupe et d'un greffon. *page 11*

Greffon (*advice*) Un greffon est un morceau de code qui implémente le comportement d'une préoccupation transverse. *page 12*

Joker (*wildcard*) Dans les motifs des expressions de point de coupe, il est possible d'utiliser deux types de joker : «⋆» qui désigne n'importe quelle une suite de caractères, et «..» qui désigne n'importe quelle suite de paramètres. *page 16*

Point de jonction (*joinpoint*) Un point de jonction est un point dans le flot d'exécution du programme où un greffon peut être tissé. *page 12*

Motif (*pattern*) Pour désigner une méthode, un constructeur, un attribut, ou une classe, une expression de point de coupe utilise un motif. Ces motifs peuvent contenir des jokers et donc désigner plusieurs éléments. *page 16*

Mutant Un mutant est une copie d'un programme dans laquelle une erreur a été introduite. *page 29*

Analyse de mutation L'analyse de mutation est une technique permettant de qualifier une suite de tests en injectant des erreurs dans un programme. *page 29*

Ombre d'un point de jonction (*joinpoint shadow*) L'ombre d'un point de jonction est le morceau de code source correspondant. Selon les points de jonction l'ombre peut être incluse dans une instruction ou être longue de plusieurs lignes de code. *page 15*

Oracle L'oracle émet un verdict lors de l'exécution d'un cas de test en vérifiant un certain nombre d'assertions. *page 27*

Expression de point de coupe (*pointcut descriptor*) Une expression de point de coupe décrit un ensemble de points de jonction dans le flot d'exécution du programme où un greffon est tissé. *page 12*

Scénario de test Le scénario de test spécifie l'état du système et le déroulement d'un cas de test. *page 27*

Cas de test (*test case*) Un cas de test exécute une partie d'un programme afin de détecter des erreurs. Il est composé d'une donnée de test, d'un oracle, et d'un scénario de test qui spécifie l'état du programme. *page 26*

Donnée de test (*test data*) La donnée de test est la donnée d'entrée du programme lors de l'exécution d'un cas de test. *page 27*

Suite de tests (*test suite*) Une suite de tests est un ensemble de cas de test ciblant un même programme ou une même unité de programme. *page 26*

Tissage (*weaving*) Le tissage est la composition qui assemble la préoccupation principale (le programme de base) et les préoccupations transverses (les aspects) pour produire le comportement global. Le tissage a lieu le plus souvent à la compilation, mais peut aussi avoir lieu à l'exécution. *page 11*

Tisseur L'outil qui effectue la composition du programme de base et des aspects. *page 11*

Table des figures

Liste des tableaux

Listings

Bibliographie

[Aldrich 05] Jonathan Aldrich. *Open Modules : Modular Reasoning About Advice*. In ECOOP'05 : Proceedings of the 19th European Conference on Object-Oriented Programming, volume 3586, pages 144–168. Springer, 2005.

[Alexander 04] Roger T. Alexander, James M. Bieman & Anneliese A. Andrews. *Towards the systematic testing of aspect-oriented programs*. Rapport technique, Colorado State University, 2004.

[Ammann 08] Paul Ammann & Jeff Offutt. Introduction to software testing. Cambridge University Press, 2008.

[Anbalagan 06] Prasanth Anbalagan & Tao Xie. *APTE : automated pointcut testing for AspectJ programs*. In WTAOP '06 : Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs, pages 27–32. ACM, 2006.

[Anbalagan 08] Prasanth Anbalagan & Tao Xie. *Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs*. In ISSRE '08 : Proceedings of the 19th International Symposium on Software Reliability Engineering, pages 239–248, 2008.

[Aracic 06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini & Klaus Ostermann. *An Overview of CaesarJ*. Transactions on Aspect-Oriented Software Development, pages 135–173, 2006.

[Bache 90] Richard Bache & Monika Mullerburg. *Measures of testability as a basis for quality assurance*. Software Engineering Journal, vol. 5, no. 2, pages 86–92, 1990.

[Bækken 06] Jon S. Bækken & Roger T. Alexander. *A Candidate Fault Model for AspectJ Pointcuts*. In ISSRE '06 : Proceedings of the 17th International Symposium on Software Reliability Engineering, pages 169–178. IEEE Computer Society, 2006.

[Barbosa 01] Ellen Francine Barbosa, José Carlos Maldonado & Auri Marcelo Rizzo Vincenzi. *Towards the determination of sufficient mutant operators fo C*. The Journal of Software Testing, Verification, and Reliability, vol. 11, no. 2, 2001.

[Baudry 05] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel & Yves Le Traon. *Automatic Test Case Optimization : A Bacteriologic Algorithm*. IEEE Software, vol. 22, no. 2, pages 76–82, 2005.

[Baudry 06] Benoit Baudry, Franck Fleurey & Yves Le Traon. *Improving test suites for efficient fault localization*. In ICSE '06 : Proceedings of the 28th International Conference on Software Engineering, pages 82–91. ACM, 2006.

[Beizer 90] Boris Beizer. Software testing techniques. Van Norstrand Reinhold, 1990.

[Beuche 06] Danilo Beuche & Cédric Beust. *AOP Has Yet to Prove Its Value*. IEEE Software, vol. 23, no. 1, pages 73–74, 2006.

[Binder 99] Robert V. Binder. Testing object-oriented systems : models, patterns, and tools. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Bonér 04] Jonas Bonér. *What are the key issues for commercial AOP use : how does AspectWerkz address them ?* In AOSD '04 : Proceedings of the 3rd international conference on Aspect-oriented Software Development, pages 5–6. ACM, 2004.

[Briand 99] Lionel C. Briand, John W. Daly & Jürgen K. Wüst. *A Unified Framework for Coupling Measurement in Object-Oriented Systems*. IEEE Transactions on Software Engineering, vol. 25, no. 1, pages 91–121, 1999.

[Bruntink 04] Magiel Bruntink & Arie van Deursen. *Predicting Class Testability using Object-Oriented Metrics*. In SCAM '04 : Proceedings of the 4th IEEE international workshop Source Code Analysis and Manipulation, pages 136–145. IEEE Computer Society, 2004.

[Bruntink 06] Magiel Bruntink & Arie van Deursen. *An empirical study into class testability*. Journal of Systems and Software, vol. 79, no. 9, pages 1219–1232, 2006.

[Cagnon 98] Etienne M. Cagnon & Laurie J. Hendren. *SableCC, an Object-Oriented Compiler Framework*. In TOOLS '98 : Proceedings of the Technology of Object-Oriented Languages and Systems, pages 140–154. IEEE Computer Society, 1998.

[Chidamber 94] Shyam R. Chidamber & Chris F. Kemerer. *A metrics suite for object oriented design*. IEEE Transactions on Software Engineering, vol. 20, no. 6, pages 476–493, 1994.

[Churcher 95] Neville I. Churcher & Martin J. Shepperd. *Towards a conceptual framework for object oriented software metrics*. IEEE Transactions on Software Engineering, vol. 20, no. 2, pages 69–75, 1995.

[Colyer 06] Adrian Colyer, Rob Harrop, Rod Johnson & Alexandre Vasseur. *AOP Will See Widespread Adoption*. IEEE Software, vol. 23, no. 1, pages 72–75, 2006.

[De Fraine 08] Bruno De Fraine, Mario Südholt & Viviane Jonckers. *StrongAspectJ : flexible and safe pointcut/advice bindings*. In AOSD '08 : Proceedings of the 7th international conference on Aspect-oriented Software Development, pages 60–71. ACM, 2008.

[Delamare 08] Romain Delamare, Benoît Baudry & Yves Le Traon. *Regression test selection when evolving software with aspects*. In LATE '08 : Proceedings of the AOSD workshop on Linking Aspect Technology and Evolution, pages 1–5. ACM, 2008.

[Delamare 09a] Romain Delamare, Benoît Baudry, Sudipto Ghosh & Yves Le Traon. *A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ*. In ICST '09 : Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation, 2009. Sélectionné pour une édition spéciale de Journal of Software Testing, Verification and Reliability (STVR).

[Delamare 09b] Romain Delamare, Benoît Baudry & Yves Le Traon. *AjMutator : A Tool For The Mutation Analysis of AspectJ Pointcut Descriptors*. In Mutation'09 : Proceedings of the 4th International Workshop on Mutation Analysis, 2009.

[DeMillo 78] Richard DeMillo, Richard J. Lipton & Frederick G. Sayward. *Hints on Test Data Selection : Help for the Practicing Programmer*. IEEE Computer, vol. 11, no. 4, pages 34–41, 1978.

[Denning 05] Peter J. Denning. *The locality principle*. Communications of the ACM, vol. 48, no. 7, pages 19–24, 2005.

[Derezinska 08] Anna Derezinska & Anna Szustek. *Tool-Supported Advanced Mutation Approach for Verification of C# Programs*. In DEPCOS-RELCOMEX '08 : Proceedings of the 2008 3rd International Conference on Dependability of Computer Systems, pages 261–268. IEEE Computer Society, 2008.

[Ferrari 08] Fabiano Cutigi Ferrari, José Carlos Maldonado & Awais Rashid. *Mutation Testing for Aspect-Oriented Programs*. In ICST '08 : Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, pages 52–61, 2008.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. Design patterns : elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., 1995.

[Griswold 06] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai & Hridesh Rajan. *Modular Software Design with Crosscutting Interfaces*. IEEE Software, vol. 23, no. 1, pages 51–60, 2006.

[Gybels 03] Kris Gybels & Johan Brichau. *Arranging language features for more robust pattern-based crosscuts*. In AOSD '03 : Proceedings of the 2nd international conference on Aspect-oriented software development, pages 60–69. ACM, 2003.

[Harman 09] Mark Harman, Fayezin Islam, Tao Xie & Stefan Wappler. *Automated test data generation for aspect-oriented programs*. In AOSD '09 : Proceedings of the 8th ACM international conference on Aspect-oriented software development, pages 185–196. ACM, 2009.

[Harrold 94] Mary Jean Harrold & Gregg Rothermel. *Performing data flow testing on classes*. In Proceedings of the 2nd symposium on Foundations of Software Engineering, pages 154–163. ACM, 1994.

[Harrold 01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon & Ashish Gujarathi. *Regression test selection for Java software*. In OOPSLA'01 : Proceedings of the 16th conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 312–326, 2001.

[Henderson-Sellers 96] B. Henderson-Sellers. Object-oriented metrics. Prentice Hall, 1996.

[Hoffman 08] Kevin Hoffman & Patrick Eugster. *Towards reusable components with aspects : an empirical study on modularity and obliviousness*. In ICSE '08 : Proceedings of the 30th international conference on Software engineering, pages 91–100. ACM, 2008.

[Kellens 06] Andy Kellens, Kim Mens, Johan Brichau & Kris Gybels. *Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts*. In 20th European Conference on Object-Oriented Programming (ECOOP 2006), 2006.

[Kiczales 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier & John Irwin. *Aspect-Oriented Programming*. In Proceedings of the European Conference on Object-Oriented Programming, 1997.

[Kiczales 01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William G. Griswold. *An Overview of AspectJ*. In ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming, pages 327–353. Springer-Verlag, 2001.

[Kiczales 05] Gregor Kiczales & Mira Mezini. *Aspect-oriented programming and modular reasoning*. In ICSE '05 : Proceedings of the 27th international conference on Software engineering, pages 49–58. ACM, 2005.

[Kim 06] Sang-Woon Kim, Mary Jean Harrold & Yong-Rae Kwon. *MUGAMMA : Mutation Analysis of Deployed Software to Increase Confidence and Assist Evolution*. In MUTATION '06 : Proceedings of the Second Workshop on Mutation Analysis, page 10. IEEE Computer Society, 2006.

[Klaeren 01] Herbert Klaeren, Elke Pulvermüller, Awais Rashid & Andreas Speck. *Aspect Composition Applying the Design by Contract Principle*. In GCSE'00 : Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering, volume 2177, pages 57–69. Springer, 2001.

[Koppen 04] Christian Koppen & Maximilian Stoerzer. *PCDiff : Attacking the fragile pointcut problem*. In European Interactive Workshop on Aspects in Software (EIWAS), 2004.

[Lagaisse 04] Bert Lagaisse, Wouter Joosen & Bart De Win. *Managing semantic interference with aspect integration contracts*. In SPLAT'04 : Software engineering Properties of Languages for Aspect, 2004.

[Lemos 06] Otávio Augusto Lazzarini Lemos, Fabiano Cutigi Ferrari, Paulo Cesar Masiero & Cristina Videira Lopes. *Testing aspect-oriented programming Pointcut Descriptors*. In WTAOP '06 : Proceedings of the 2nd workshop on Testing aspect-oriented programs, pages 33–38. ACM, 2006.

[Lemos 07] Otávio Augusto Lazzarini Lemos, Auri Marcelo Rizzo Vincenzi, José Carlos Maldonado & Paulo Cesar Masiero. *Control and data flow structural testing criteria for aspect-oriented programs*. Journal of Systems and Software, vol. 80, no. 6, pages 862–882, 2007.

[Li 93] Wei Li & Sallie Henry. *Object-oriented metrics that predict maintainability*. Journal of Systems and Software, vol. 23, no. 2, pages 111–122, 1993.

[Lopes 05] Cristina Vidieira Lopes & Trung Chi Ngo. *Unit testing aspectual behavior*. In Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs, 2005.

[Lopez-Herrejon 06] Roberto Lopez-Herrejon, Don Batory & Christian Lengauer. *A disciplined approach to aspect composition*. In PEPM '06 : Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 68–77. ACM, 2006.

[Ma 02] Yu-Seung Ma, Yong-Rae Kwon & Jeff Offutt. *Inter-Class Mutation Operators for Java*. In ISSRE '02 : Proceedings of the 13th International Symposium on Software Reliability Engineering, page 352. IEEE Computer Society, 2002.

[Ma 05] Yu-Seung Ma, Jeff Offutt & Yong-Rae Kwon. *MuJava : an automated class mutation system*. The Journal of Software Testing, Verification, and Reliability, vol. 15, no. 2, pages 97–133, 2005.

[McEachen 05] Nathan McEachen & Roger T. Alexander. *Distributing classes with woven concerns : an exploration of potential fault scenarios*. In AOSD '05 : Proceedings of the 4th international conference on Aspect-oriented software development, pages 192–200. ACM, 2005.

[Meyer 92] Bertrand Meyer. *Applying "Design by Contract"*. Computer, vol. 25, no. 10, pages 40–51, 1992.

[Munoz 07] Freddy Munoz, Olivier Barais & Benoît Baudry. *Vigilant Usage of Aspects*. In Proceedings of the 2nd International Workshop on Aspects, Dependencies and Interactions associated with ECOOP'07, 2007.

[Munoz 08] Freddy Munoz, Benoit Baudry & Olivier Barais. *Improving maintenance in AOP through an interaction specification framework*. In ICSM'09 : Proceedings of the 24th International Conference on Software Maintenance, 2008.

[Munoz 09] Freddy Munoz, Benoit Baudry, Romain Delamare & Yves Le Traon. *Inquiring the usage of aspect-oriented programming : an empirical study*. In ICSM '09 : Proceedings of the 25th International Conference on Software Maintenance, 2009.

[Myers 79] Glenford J. Myers. The art of software testing. John Wiley & Sons, Inc., 1979.

[NIST 02] NIST. *Software Errors Cost U.S. Economy $59.5 Billion Annually*, 2002. `http://www.nist.gov/public_affairs/releases/n02-10.htm`.

[Offutt 92] A. Jefferson Offutt. *Investigations of the software testing coupling effect*. Transactions on Software Engineering and Methodology, vol. 1, no. 1, pages 5–20, 1992.

[Offutt 96] A. Jefferson Offutt, Jeff Voas & Jeff Payne. *Mutation Operators for Ada*. Rapport technique, 1996.

[Offutt 97] Jeff Offutt & Jie Pan. *Automatically Detecting Equivalent Mutants and Infeasible Paths*. The Journal of Software Testing, Verification, and Reliability, vol. 7, no. 3, pages 165–192, 1997.

[Ostermann 05] Klaus Ostermann, Mira Mezini & Cristoph Bockisch. *Expressive Pointcuts for Increased Modularity*. In ECOOP'05 : Proceedings of the 19th European Conference on Object-Oriented Programming, volume 3586, pages 214–240. Springer, 2005.

[Ostrand 88] T. J. Ostrand & M. J. Balcer. *The category-partition method for specifying and generating functional tests*. Communications of the ACM, vol. 31, no. 6, pages 676–686, 1988.

[Pawlak 04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry & Laurent Martelli. *JAC : an aspect-based distributed dynamic framework*. Softw. Pract. Exper., vol. 34, no. 12, pages 1119–1148, 2004.

[Pawlak 06] Renaud Pawlak, Carlos Noguera & Nicolas Petitprez. *Spoon : Program Analysis and Transformation in Java*. Rapport technique 5901, INRIA, 2006. http ://spoon.gforge.inria.fr.

[Pretschner 05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch & T. Stauner. *One evaluation of model-based testing and its automation*. In ICSE '05 : Proceedings of the 27th international conference on Software engineering, pages 392–401. ACM, 2005.

[Rinard 04] Martin Rinard, Alexandru Salcianu & Suhabe Bugrara. *A classification system and analysis for aspect-oriented programs*. In FSE'04 : Proceedings of the 12th international symposium on Foundations of Software Engineering, pages 147–158, 2004.

[Rothermel 96] Gregg Rothermel & Mary Jean Harrold. *Analyzing regression test selection techniques*. IEEE Transactions on Software Engineering, vol. 22, no. 8, pages 529–551, 1996.

[Rothermel 97] Gregg Rothermel & Mary Jean Harrold. *A safe, efficient regression test selection technique*. ACM Trans. Softw. Eng. Methodol., vol. 6, no. 2, pages 173–210, 1997.

[Sakurai 08] Kouhei Sakurai & Hidehiko Masuhara. *Test-based pointcuts for robust and fine-grained join point specification*. In AOSD '08 : Proceedings of the 7th international conference on Aspect-oriented software development, pages 96–107. ACM, 2008.

[Skotiniotis 04] Therapon Skotiniotis & David H. Lorenz. *Cona : aspects for contracts and contracts for aspects*. In OOPSLA '04 : Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 196–197. ACM, 2004.

[Steimann 04] Friedrich Steimann. *Aspects are technical, and they are few*. In European Interactive Workshop on Aspects in Software EIWAS'04, 2004.

[Steimann 06] Friedrich Steimann. *The paradoxical success of aspect-oriented programming*. In OOPSLA '06 : Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming

systems, languages, and applications, pages 481–497. ACM, 2006.

[Stoerzer 05] Maximilian Stoerzer & Juergen Graf. *Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software*. In ICSM '05 : Proceedings of the 21st IEEE International Conference on Software Maintenance, pages 653–656. IEEE Computer Society, 2005.

[Utting 06] Mark Utting & Bruno Legeard. Practical model-based testing - a tools approach. Elsevier Science, 2006. 550 pages, ISBN 0-12-372501-1.

[van der Werff 01] Terry J. van der Werff. *10 Emerging Technologies That Will Change the World*. Technology Review, 2001.

[Wedyan 08] Fadi Wedyan & Sudipto Ghosh. *A Joinpoint Coverage Measurement Tool for Evaluating the Effectiveness of Test Inputs for AspectJ Programs*. In ISSRE '08 : Proceedings of the 2008 19th International Symposium on Software Reliability Engineering, pages 207–212. IEEE Computer Society, 2008.

[Xie 06a] Tao Xie & Jianjun Zhao. *A framework and tool supports for generating test inputs of AspectJ programs*. In AOSD'06 : Proceedings of the 5th international conference on Aspect-Oriented Software Development, pages 190–201, 2006.

[Xie 06b] Tao Xie, Jianjun Zhao, Darko Marinov & David Notkin. *Detecting Redundant Unit Tests for AspectJ Programs*. In Proceedings of the 17th International Symposium on Software Reliability and Engineering, pages 179–190, 2006.

[Xu 05] Dianxiang Xu, Weifeng Xu & Kendall Nygard. *A state-based approach to testing aspect-oriented programs*. In Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, 2005.

[Xu 06] Dianxiang Xu & Weifeng Xu. *State-based incremental testing of aspect-oriented programs*. In Proceedings of the 5th International conference on Aspect-Oriented Software Development, pages 180–189, 2006.

[Xu 07] Guoqing Xu & Atanas Rountev. *Regression Test Selection for AspectJ Software*. In ICSE '07 : Proceedings of the 29th International Conference on Software Engineering, pages 65–74, 2007.

[Ye 08] Lingdong Ye & Kris De Volder. *Tool support for understanding and diagnosing pointcut expressions*. In AOSD '08 : Proceedings of the 7th international conference on Aspect-oriented Software Development, pages 144–155. ACM, 2008.

[Zhang 08] Sai Zhang, Zhongxian Gu, Yu Lin & Jianjun Zhao. *Celadon : a change impact analysis tool for aspect-oriented programs*. In ICSE Companion '08 : Companion of the 30th international conference on Software engineering, pages 913–914. ACM, 2008.

[Zhao 03] Jianjun Zhao. *Data-Flow-Based Unit Testing of Aspect-Oriented Programs*. In Proceedings of the 27th Annual International Conference on Computer Software and Applications, page 188. IEEE Computer Society, 2003.