Impacts de l'AOP sur les tests dans un environnement Agile

Utilisation de Mocks pour les tests unitaires d'aspects

Mémoire

Félix-Antoine Bourbonnais-Bigras

Maîtrise en informatique
Maître ès sciences (M.Sc.)

Québec, Canada

Résumé

Bien que l'AOP soit présent dans le paysage informatique depuis quelques années, son adoption industrielle reste relativement timide malgré les bénéfices architecturaux promis et espérés.

L'objectif principal de nos travaux est de favoriser l'adoption de l'AOP auprès des développeurs oeuvrant dans l'industrie. Nous voulons proposer des solutions adaptées qui permettent à ces professionnels d'embrasser l'AOP sans avoir à sacrifier leur processus, méthodologie et outils actuels qui leur permettent d'atteindre des objectifs de performance, de qualité, de déploiement et de maintenabilité.

Afin de nous permettre d'identifier des besoins réels, nous avons centré nos travaux sur des équipes employant un processus Agile. En effet, les processus Agiles préconisent généralement diverses pratiques et principes qui risquent d'être affectés par un changement de paradigme de programmation. C'est notamment le cas des tests qui tiennent une place très importante au sein des méthodes agiles, car ils permettent l'acceptation du changement sur le long terme.

Dans un premier temps, nous avons mené une étude exploratoire dont l'objectif était de mieux définir quels seraient les irritants pour des développeurs débutants avec l'AOP et qui travaillent dans un environnement Agile.

À la lumière des résultats de cette étude, nous avons amorcé la partie maîtresse de nos travaux qui consistent à élaborer une solution pour faciliter les tests unitaires d'aspects. Pour ce faire, nous avons conçu et rendu disponible un cadre d'applications (framework) permettant la création et le tissage de Mocks avec des aspects, facilitant ainsi l'isolation des aspects. Ce cadre a été développé de manière à s'arrimer avec les pratiques et outils répandus en industrie pour des équipes Agiles.

Abstract

Although the fact that AOP has been proposed since quite a few years, the industrial adoption of this paradigm is not widespread in spite of the benefits hoped and promised for software architectures.

The main goal of our work is to promote the adoption of AOP by industrial developers. We want to provide adapted solutions allowing those professionals to embrace AOP without having to sacrifice their processes, practices and tools. We seek practical solutions to allow them to reach their performance, quality, deployment and maintainability goals.

To better identify and understand the real needs of teams using Agile processes, our work is focus on Agile environments. In fact, Agile processes encourage numerous practices and principles that could be affected by the usage of AOP in such projects. For instance, this is the case for software testing who is largely embraced the Agile community as a way of sustaining the acceptation of change over time.

Our first step was to conduct an exploratory study to identify what could be the irritants for Agile teams who want to use AOP for the first time.

Considering those results, we undertook our main contribution consisting to produce a solution to help unit testing of aspects. To do so, we developed and made available a framework to create and weave Mocks with aspects to help isolate aspects for testing purposes. The framework was designed to integrate easily with practices and tools commonly used in the industry by Agile teams.

Table des matières

Liste des figures

Remerciements

Cela fait maintenant presque huit ans que je connais celui qui deviendra plus tard mon directeur, Luc Lamontagne. Luc est une personne curieuse, passionnée, humble et d'une grande ouverture d'esprit. Il m'est impossible d'estimer tous ces moments que nous avons passés à discuter ou simplement jaser sans trop nous apercevoir du temps qui filait. J'ai eu la chance d'avoir un directeur qui s'intéresse à tout et qui partage la même passion que moi pour le génie logiciel. Merci Luc pour tes conseils, mais, surtout, pour ta patience et ta compréhension alors que j'entreprenais trop de projets en même temps ce qui, inévitablement, retardait la rédaction de cette maîtrise.

Merci à Claude Bolduc pour ses précieux conseils, ses commentaires et son support. Claude a toujours été là pour m'aider et pour me guider dans les méandres de la recherche universitaire. Merci pour ton mentorat et pour m'avoir forcé à me focaliser sur mes objectifs.

Côté technique, personne ne peut battre les conseils de Georges-Etienne Legendre. En plus de son oeil de lynx, Georges-Etienne est un professionnel doté d'un pragmatisme avisé dont les opinions valent leur pesant d'or. Merci d'avoir accepté d'écouter mes problèmes incompréhensibles pendant des heures et d'avoir révisé tout mon code.

Merci à Marc Fluet pour ses conseils et le coaching lors de ses passages à Québec. Malgré que nos domaines ne soient pas les mêmes, tes conseils étaient toujours pertinents et m'ont grandement aidé.

Un immense merci à mes parents et Xu pour leur support, leur confiance, leurs opinions et leurs révisions.

En terminant, je voudrais également remercier Jean-François Roy pour avoir révisé mes publications anglophones. Je ne voudrais pas oublier Abder Alikacem sans qui je n'aurais peut-être jamais fait d'études de deuxième cycle. Abder m'a donné ma chance alors que j'étais jeune bachelier et je lui en serai toujours extrêmement reconnaissant.

Sur une dernière note, merci à tous mes étudiants qui ont suivi mon cours dans les dernières années pour m'avoir forcé à me tenir à jour et à continuellement me remettre en question.

Chapitre 1

Introduction

Depuis les débuts de l'orientation objet, il y a maintenant plus de trente ans, les techniques, outils mais également les problèmes et leurs contraintes ont grandement changés. Les systèmes informatiques sont de plus en plus grands, sont disponibles au grand public via Internet et sont même souvent géo-délocalisés.

Cette évolution et en particulier l'ampleur des systèmes ont forcé le développement de techniques et pratiques architecturales visant à réduire les imposants problèmes de maintenance. L'ajout de fonctionnalités ainsi que la modification des systèmes ont, en effet, donné lieu à de nombreuses histoires effarantes de dépassements des coûts, des logiciels impossibles à faire évoluer dans des délais raisonnables ou encore à des taux de défectuosités atteignant des sommets.

Cette nouvelle réalité n'a pas seulement forcé les communautés à proposer des solutions architecturales mais également à se pencher sur le processus même de fabrication des logiciels. Processus destinés à produire des logiciels plus facilement maintenables et répondant plus adéquatement aux besoins, souvent changeants, des clients.

L'AOP de même que les processus Agiles sont des innovations s'inscrivant dans cette nouvelle réalité et qui ont gagné une certaine popularité durant dans la dernière décennie.

Provenant de deux disciplines différentes du génie logiciel et se situant à des niveaux distincts, ces deux concepts ne sont pas conflictuels et ont, bien au contraire, un fort potentiel d'être utilisés conjointement. En ce sens, Agile peut offrir un cadre s'appliquant au cycle de vie du logiciel alors que l'AOP offre une solution architecturale à un niveau plus technique et conceptuel.

Cependant, ces deux solutions proviennent de courants de pensées différents et, surtout, n'ont pas été conçues spécifiquement de manière à s'appliquer conjointement. Car, il arrive parfois que certaines solutions puissent être développées en ayant en tête une utilisation conjointe comme RUP et UML, par exemple. Or, le mouvement Agile étant un courant de pensée plutôt marginal à l'époque de la création de l'AOP, les outils et implémentations en découlant, tel

AspectJ, n'ont pas forcément inclus les outils nécessaires afin d'offrir un support maximal aux pratiques Agiles comme, par exemple, le réusinage (refactoring).

Les travaux présentés dans le cadre de ce mémoire ont, par conséquent, pour objectif de faciliter l'arrimage de l'AOP lorsque utilisés dans le cadre d'un projet Agile. Arrimage permettant aux équipes Agiles de bénéficier pleinement des avantages combinés des deux technologies. Pour ce faire, nous débuterons par une amorce d'analyse avec une étude pilote (chapitre 3) ; puis nous proposerons un outil permettant aux développeurs d'effectuer des tests unitaires d'aspects.

1.1 AOP

La programmation orientée aspect [1] (AOP ou POA) est un paradigme de programmation ayant fait son apparition il y a maintenant un peu plus de dix ans mais dont l'adoption, bien que de plus en plus répandue, semble encore relativement timide par rapport à l'orientation objet (OO) quasi omniprésente dans l'industrie.

L'un des objectifs de la programmation orientée objet (OOP) était de favoriser la modularité et l'encapsulation via le concept des objets. Cela dans le but de faciliter la maintenabilité des logiciels grâce au regroupement des comportements similaires et en limitant leur dispersion, voire leur duplication.

Or, il subsiste une classe de problèmes récalcitrants que l'OO ne parvient pas à correctement regrouper et qui ont tendance à disperser et contaminer le code. C'est pour cette raison que l'on qualifie ces *préoccupations* de *transverses* ou *cross-cutting concerns* en anglais (voir section 2.3.1).

L'AOP a émergé dans le but d'offrir une meilleure prise en charge des *préoccupations transverses*. Pour cette raison, l'AOP est souvent grossièrement présenté comme une évolution de l'OOP puisqu'il s'agit d'un paradigme repoussant les limites de l'OO tout en poursuivant, grosso modo, les mêmes idéaux.

De plus amples informations concernant l'AOP seront présentées à la section 2.3.

1.2 Agile

Très tôt, les entreprises de fabricants du logiciel ont constaté le besoin d'encadrer le développement logiciel dans des processus plus ou moins formels offrant des balises, des pratiques et une structure afin de les guider tout au long du cycle de vie du logiciel.

Ces processus logiciels, souvent appelés méthodes ou méthodologies, peuvent varier grandement mais couvrent généralement la planification, l'organisation du travail ainsi que l'ordre et

1. Dans la suite de ce mémoire, nous désigneront la programmation orientée aspect par son acronyme anglais *AOP* pour *Aspect Oriented Programming*. Il en sera de même pour les termes dérivés tels que *AO* pour *Aspect Orientation*.

parfois la manière dont les tâches et étapes sont réalisées dans le but de produire un logiciel répondant, le plus possible, au besoin du client dans les délais et au moindre coût.

Rapidement, est apparu le modèle en cascade (waterfall) qui est, grossièrement, un mode séquentiel où les différentes étapes sont réalisées les unes après les autres avec peu, voire pas, de retour en arrière. Cette vision était principalement basée sur les autres domaines du génie. Des variantes moins contraignantes ont cependant rapidement émergé. Avec la venue de l'OO, des processus centrés sur ce paradigme ont également vu le jour comme UP dont RUP est probablement l'implémention la plus célèbre.

C'est, entre autres, en réaction à ces modèles très rigides s'appuyant sur une planification stricte que le mouvement Agile est né au tournant du millénaire[KR02]. Le mouvement Agile, bien qu'ayant débuté avant, s'est concrétisé en 2001 par la signature du manifeste Agile [BBvB+01].

Les processus de la famille Agile sont des processus incrémentaux et itératifs qui ont, notamment, les caractéristiques suivantes :

- acceptation du changement ;
- forte adaptation ;
- forte implication du client ;
- plus focalisé sur les personnes que sur le processus ;
- accorde une plus grande importance à un logiciel fonctionnel qu'à la documentation ;
- prône l'auto-organisation des équipes ;
- etc.

Pour atteindre ces objectifs une série de pratiques et principes sont mis en place par les équipes pratiquant l'Agilité. Agile n'est pas un processus en tant que tel mais représente plutôt une famille de processus ayant des valeurs communes. Il existe donc, par conséquent, plusieurs processus Agiles dont eXtreme Programming, Scrum, Lean, etc.

Agile étant né d'un mouvement réactionnaire mais également en raison du gain extrêmement rapide de popularité de ces processus, Agile est parfois devenu une jargonnerie [2] pour caractériser des processus qui ne répondent pourtant pas aux valeurs des processus Agiles.

Agile est une vaste famille et il convient d'être prudent car il existe plusieurs mythes et préjugés concernant Agile. Ainsi, les processus Agiles ont très souvent été associés, à tord, à l'absence de processus, au laissez-faire ou au chaos [KR02].

Voir section 2.4 pour de plus amples informations concernant les processus Agiles.

2. jargonnerie (en : buzzword) : Mot d'un jargon spécialisé utilisé principalement pour impressionner les non-initiés ou pour donner l'impression que l'on connaît bien le domaine en question (source : OQLF).

1.3 Motivations

Impact du processus logiciel sur le développement

Les processus Agiles et l'AOP opèrent à des niveaux différents puisque l'un est un processus alors que le second est un paradigme de programmation. Or, le développement logiciel est une activité qui se déroule et qui est encadrée par un processus logiciel. Par conséquent, il est fort possible que l'AOP soit employé dans le cadre d'activités de développement à l'intérieur d'un processus Agile.

En théorie, le processus logiciel est étranger aux paradigmes utilisés lors du développement. Ainsi, un processus Agile peut être utilisé pour la réalisation de logiciels procéduraux, OO ou AOP et ce, dans n'importe quel langage de programmation.

Cependant, certains principes d'Agile demandent la mise en place de pratiques concrètes impactant directement le développement [3]. C'est le cas, notamment, du réusinage ou des tests qui demandent aux développeurs d'adapter leur manière de développer au quotidien.

De plus, le processus impacte également les artefacts produits lors du développement. Par exemple, un processus Agile aura tendance à produire beaucoup de tests automatisés mais moins de documentation externe au code.

Finalement, le processus agira également sur le moment où ces artefacts sont produits ainsi que leur état d'avancement à chacune de ces étapes. Dans un processus traditionnel, le design et l'architecture sont généralement amenés dans un état quasi-définitif avant de débuter le code. Même dans des processus plus itératifs comme RUP, passé un certain point l'architecture sera dans un état assez stable et changera peu ; bien qu'il ne soit pas défendu d'y revenir. En Agile, l'amélioration du design et de l'architecture est presque permanente via la pratique de réusinage (refactoring) tout au long du projet [Fow04a]. De plus, les tests s'ajouteront généralement au même rythme que le code et ce, dès le début et pour toute la durée du projet.

Impacts de l'AOP dans un projet Agile

Pour ces raisons, les contraintes et les agréments offerts par un paradigme peuvent parfois favoriser ou nuire à certaines pratiques recommandées par le processus logiciel employé.

Dans le cas de l'AOP, rappelons que le choix de ce paradigme offre des avantages principalement architecturaux. En ce sens, nous pouvons émettre comme hypothèse que l'AOP pourrait favoriser certaines pratiques ou principes Agiles comme :

- le réusinage en permettant une plus grande liberté en terme de modularité ;
- le réduction du coût du changement dans le temps puisque l'élimination des préoccupations transverses vise à faciliter la maintenabilité à long terme (voir sections 2.1 et 2.3).

3. La liste des pratiques varie en fonction du processus Agile utilisé (XP, Scrum, etc.). Cependant, il existe un ensemble de pratiques communes à la plupart des processus et qui sont presque de facto acceptées par la communauté Agile internationale.

Pour plus de détails, voir la section 3 concernant les hypothèses de recherche.

Étant donné que Agile prône la qualité et l'amélioration continue du design, il pourrait être logique de penser que l'AOP n'a que des effets positifs sur les pratiques de développement d'Agile.

Or, il est possible d'entrevoir comme hypothèse deux cas où l'utilisation de l'AOP pourrait parfois contredire ou nuire à des principes ou pratiques Agiles :

- L'ajout d'aspects et de l'AOP peut rendre le design plus complexe (en raison de l'abstraction, du renversement du flot, etc.). L'augmentation potentielle de la complexité pourrait rendre les ajouts et les modifications plus onéreux et plus enclins à générer des bogues.

  Or, la philosophie Agile repose sur le fait que le coût des ajouts et modifications est maintenu le plus bas possible tout au long du projet afin d'être en mesure d'accepter les changements sans trop de heurts.

- Le manifeste Agile [BBvB+01] érige la « simplicité » [4] comme étant un principe crucial. Or, l'introduction d'aspects peut ne pas être la solution la plus simple afin d'améliorer le design lors d'un réusinage.

  Il peut arriver, par exemple, que pour certaines préoccupations transverses simples et peu envahissantes, l'introduction du Design Pattern soit un compromis suffisant dans l'équilibre simplicité versus maintenabilité.

Impacts de l'AOP sur les équipes Agiles

Dans la recherche de ce difficile équilibre entre la simplicité, la maintenabilité et la productivité, il faut également considérer le niveau de familiarité de l'équipe avec les solutions possibles. Ainsi, l'introduction d'un aspect par une équipe peu expérimentée avec l'AOP sera, avec raison, considérée comme plus complexe et plus difficile à maintenir.

Expériences antérieures de l'équipe

Il en va de même avec la fréquence d'utilisation de l'AOP dans le projet courant. Même pour une équipe très familière avec AspectJ, le fait d'introduire le premier aspect dans un projet aura un coût de maintenance plus élevé par rapport à l'introduction du même aspect dans un projet en comportant déjà plusieurs autres.

Ceci s'explique par le fait qu'introduire un premier aspect en AspectJ dans un projet revient à introduire une nouvelle technologie. Ceci pourrait, par exemple, demander des modifications à l'environnement de développement de l'équipe, au système de déploiement, aux tests automatisés, etc.

4. Le manifeste définit la simplicité comme étant l'art de maximiser la quantité de travail à ne *pas* faire. [BBvB+01]

Considérations externes au développement

Ces modifications à l'environnement ont, non seulement un coût mais risquent également de complexifier la gestion de cet environnement. Dans certaines entreprises, par exemple, les techniciens responsables de l'installation ou du support devront être formés en conséquence. Même pour les développeurs, identifier les problèmes ou s'assurer du déploiement sur des serveurs applicatifs (ex. : tissage LTW [5] dans un conteneur Java EE [6] avec AspectJ) peut devenir plus complexe.

L'équipe doit alors, comme souvent, balancer les avantages et inconvénients d'une technologie en considérant les impacts sur l'ensemble du processus et non seulement sur l'architecture logicielle. Il leur faut donc considérer les impacts, non seulement, sur la maintenance du code mais également sur l'ensemble des systèmes et de l'infrastructure gravitant autour du logiciel.

La section 2.6.2 présente des exemples d'impacts potentiels de l'utilisation de l'AOP sur l'environnement en fonction de certaines technologies employées.

Impacts sur l'entreprise et les humains

Malheureusement, la majorité des études sont centrées sur le cycle de développement, voire focalisées sur certains aspects architecturaux précis (voir chapitres 2 et 3). Or, le développement logiciel ne se déroule pas en vase clos mais s'inscrit plutôt dans un contexte d'ingénierie plus vaste impliquant des processus, des *contraintes* économiques, des politiques d'entreprises, mais également des *humains*.

Ces entreprises et leurs développeurs humains ont des *façons de faire*, souvent éprouvées, des *outils*, des *habitudes* mais, surtout, des *réactions et des émotions humaines* qu'il faut considérer.

Importance des outils et de l'environnement de développement

De nos jours, les équipes de développement sont de plus en plus grandes et les projets de plus en plus imposants. Cette nouvelle réalité demande maintenant aux équipes de se doter d'outils afin de mieux se coordonner et de maximiser l'automatisation.

De plus, la taille des projets requiert également l'emploi de technologies variées au sein d'un même projet. Ainsi, il n'est pas rare de voir cohabiter plusieurs langages de programmation et plusieurs composantes répondants à des besoins très différents.

Cette hétérogénéité requiert également la mise en place d'une infrastructure imposante et complexe. Il est donc primordial pour les équipes de rendre la manipulation et l'utilisation de cette infrastructure technique le plus simple, automatisé mais surtout *intégré* avec le développement.

5. Load-Time Weaving.
6. Java Platform Enterprise Edition anciennement appelé J2EE.

1.4 Objectifs

1.4.1 Objectifs préalables

Notre objectif général est d'éventuellement favoriser l'adoption de l'AOP par des entreprises dont les projets se déroulent en mode Agile. Pour ce faire, nous tenterons d'identifier des irritants pour ensuite proposer une solution qui pourrait en amoindrir les effets.

Dans un premier temps, notre objectif sera d'ouvrir le champ pour une meilleure *compréhension* des *problèmes que pourraient vivre les équipes* qui voudraient utiliser *l'AOP* dans le contexte d'un projet *Agile*. Nous nous intéressons particulièrement à des jeunes développeurs qui n'ont aucune connaissance antérieure de l'AOP.

Nous n'avons pas comme objectif d'identifier hors de tout doute ces problèmes mais plutôt d'augmenter notre confiance envers le fait que la solution que nous proposerons répondra à un problème réel et vécu. Il s'agira d'une exploration initiale afin de nous permettre de choisir plus judicieusement un problème à aborder plutôt que de s'appuyer uniquement sur nos propres hypothèses concernant les besoins industriels.

Ensuite, nous proposerons une solution qui permet de réduire un irritant majeur vécu par les équipes désirant adopter l'AOP dans un contexte Agile. Nous choisirons un problème consensuel et qui est *fortement conflictuel avec les principes et valeurs de l'Agilité*.

1.4.2 Objectifs suite à l'étude

Suite à l'étude (chapitre 3), nous avons choisi d'attaquer le problème des tests unitaires d'aspects.

Notre objectif sera donc de concevoir un cadre applicatif (framework) qui permettra de faciliter les tests unitaires d'aspects (en isolation, section 2.5.2).

Le but de nos recherches n'est pas de développer un outil de vérification formel permettant de prouver les spécifications d'un aspect. Nous souhaitons plutôt, à ce stade, fournir à l'industrie un moyen de tester, de manière toute aussi efficace leurs systèmes Java (OOP) et AspectJ (AOP) dans un contexte industriel complexe impliquant de nombreuses technologies et méthodologies.

Les objectifs spécifiques concernant le projet (Mock4Aj) sont énoncés et détaillés à la section 4.1.

1.5 Méthodologie

Avant de parvenir à proposer une solution permettant de mitiger au moins un irritant, nous voulons avoir une meilleure idée des irritants principaux. Pour ce faire nous réalisons, en premier lieu, une étude exploratoire dont l'objectif est de déterminer sur quel aspect nous concentrer par la suite.

Une fois cet irritant identifié et le problème cerné, nous concevons une solution pouvant mitiger ce problème. Nous développons cette solution itérativement et incrémentalement en adoptant des pratiques Agiles. Nous adoptons également des pratiques de qualité, d'utilisabilité et architecturales afin de nous assurer de développer un produit (et non simplement un prototype) qui pourra continuer à évoluer et servir à l'industrie.

C'est également pour cette raison que nous rendons ce produit disponible gratuitement et en libérons le code source (Open Source sous licence LGPL).

1.6 Structure du mémoire

Le chapitre 2 exposera une série de concepts et hypothèses préalables sur lesquels repose notre recherche et notre solution. Le chapitre 3 traitera de l'étude exploratoire que nous avons réalisée et en exposera les résultats.

Les chapitres 4 et 5 présentent notre solution au problème des tests unitaires d'aspects avec AspectJ. Le premier (chapitre 4) posera le problème, les hypothèses, nos objectifs et notre démarche. Nous discuterons également des avantages et limites de la solution que nous proposons. Ensuite, le chapitre 5 détaillera l'implémentation de la solution. Nous expliquerons comment fonctionne et est conçu le cadre applicatif Mock4Aj que nous avons créé.

Finalement, le chapitre 6 sera l'occasion d'un bref retour sur nos objectifs ainsi que les travaux futurs qui pourraient être réalisés suite à notre projet.

Chapitre 2

Notions préalables

2.1 Maintenabilité

En génie logiciel, l'ajout ou la modification du code est un problème récurrent. En effet, les logiciels doivent pouvoir évoluer et être maintenus souvent sur de longues périodes et les ajouts ou modifications n'ont souvent pas été prévus à l'origine du projet.

Chaque modification a un coût que l'on nomme le *coût du changement* et a tendance à augmenter significativement dans le temps. Afin de réduire le coût de ces changements, il existe une série de bonnes pratiques, principes et patrons.

Cette capacité de facilement garder à jour, adapter mais également modifier un logiciel tout au long de son cycle de vie porte le nom de *maintenabilité*.

Gestion des dépendances En architecture logicielle, plusieurs pratiques, principes, patrons et styles visent à répondre aux problèmes que pose la gestion des dépendances entre les modules. Ce problème peut paraître simple mais est, et a presque toujours été, au centre des préoccupations des concepteurs de logiciels. Il s'agit, en fait, d'un facteur prépondérant dans la maintenabilité d'un logiciel autant pour le coût des modifications que pour le risque reliés à ces dernières.

Lorsqu'une modification est apportée à un logiciel, on désire généralement

- limiter le nombre de modules qui seront impactés (directement ou indirectement) ;
- concentrer les modifications nécessaires à des endroits prévisibles.

Le paradigme orienté objet permet de faciliter la gestion des dépendances via l'encapsulation et d'autres principes comme nous le verrons à la section 2.2. Cependant, nous constaterons également les limites de l'orientation objet dans ce domaine et comment l'orientation aspect peut venir en renfort (section 2.3).

2.2 Programmation orientée objet

La programmation orientée objet (POO ou OOP) est un paradigme de programmation extrêmement répandu et largement utilisé de nos jours. Malgré son omniprésence dans les cursus académiques, il n'existe pas de consensus sur une définition formelle. Malgré l'absence d'une définition unique, on peut néanmoins recenser un certain nombre de concepts et éléments fondamentaux largement répandus dans la littérature [Arm06] [1].

Encapsulation La majorité des définitions [Arm06] incluent généralement le concept d'encapsulation. Formellement, l'encapsulation consiste à regrouper les constituants et les détails internes d'une abstraction en rendant ces derniers inaccessibles aux autres objets [Soc04, (SWEBOK)].

Dans la pratique, l'idée d'encapsulation prônée par l'orientation objet (OO) revient à regrouper les comportements et états d'un objet de manière à n'exposer que les comportements tout en cachant les détails internes d'implémentation (fonctionnement, données, autres objets utilisés, etc.) au monde extérieur. Les comportements opéreront alors sur les données, qui sont cachées, avec lesquelles ils sont regroupés.

On dit souvent qu'en OO, il faut exposer le *« quoi »* tout en cachant le *« comment »*.

L'encapsulation permet de réduire, mais surtout de circonscrire, l'impact en cas de modifications aux données ou à la manière dont les actions sont réalisées. En effet, le fait que les objets se comportent comme des boîtes noires, permet de changer complètement la manière dont l'objet accomplit ses tâches sans pour autant impacter ses utilisateurs.

Dépendances L'essence et la nouveauté apportées par l'OO tiennent probablement beaucoup dans la manière de gérer les dépendances. Il a été désormais beaucoup plus facile de minimiser l'impact des modifications par un meilleur contrôle et une meilleure compréhension du couplage entre des objets. Cela combiné à l'abstraction d'objets, permet de concevoir plus facilement des logiciels plus aisément maintenables. Cela a ouvert la porte aux principes et pratiques de maintenabilité et de gestion des dépendances comme les principes S.O.L.I.D. [Mar02a]

1. Cette liste est basée sur la fréquence d'apparition des concepts dans la littérature entre 1966 et 2005.

2.3 Programmation orientée aspect (AOP)

Dans la section précédente, nous avons vu qu'une des façons d'augmenter la maintenabilité est de limiter la portée des modifications nécessaires afin de faire un ajout ou une modification au logiciel. Pour ce faire, l'OO prône le groupement des concepts (cohésion) et la réduction des dépendances (couplage).

Cependant, nous verrons dans la présente section qu'il existe certaines fonctionnalités que l'OO, seule, ne permet pas de regrouper adéquatement et pour lesquelles le coût et les problèmes de maintenance restent élevés.

2.3.1 Préoccupations et responsabilités

Responsabilité unique (SRP)

Idéalement, chaque classe, et même chaque méthode, ne devrait avoir qu'un seul but ou préoccupation. Ainsi, une méthode ne devrait faire qu'une et une seule chose. Ce principe, appelé *Single Responsibility Principle (SRP)* [Mar02c], est un élément fondamental afin de bâtir des systèmes plus facilement maintenables et réutilisables. En d'autres mots, on dit qu'une classe ne devrait avoir qu'une seule *préoccupation*.

> **Note 1**
>
> Le principe du SRP a d'abord été présenté par Tom DeMarco et Meilir Page-Jones [Pag88, Dem79] simplement sous le nom général de « cohésion ».
>
> Il fut, par la suite, popularisé par Robert C. Martin dans [Mar02c] et repris dans plusieurs des ses ouvrages [Mar02a, Mar08a]. Ce principe constitue, d'ailleurs, le *« S »* de l'acronyme *« S.O.L.I.D »*, une célèbre liste de principes OO fondamentaux.
>
> Notons que R. C. Martin énonce le principe dans une formulation plus stricte : « there should never be more than one reason for a class to change [Mar02c] ».

Préoccupations primaires et secondaires

Étant donné la nature de l'orientation objet, les responsabilités des classes devraient être, le plus possible, liées à la logique d'affaires (métier) modélisée par le système.

Or, tout système possède des préoccupations techniques qui supportent la logique d'affaires sans y contribuer directement. On nomme ces préoccupations *secondaires* par opposition aux préoccupations *primaires* qui elles, répondent directement aux besoins fondamentaux et fonctionnels du logiciel.

Exemple 1

Dans un système bancaire, on peut imaginer avoir des préoccupations primaires telles que la gestion des comptes ou des flux monétaires. Par contre, une préoccupation comme l'intégrité des transactions en base de données, bien qu'extrêmement importante, serait probablement qualifiée de secondaire. Il s'agirait alors d'un besoin technique supportant la logique d'affaires.

Notons, que le fait qu'une préoccupation soit classée dans la catégorie secondaire, ne la rend pas moins importante qu'une préoccupation primaire. Cependant, nous nous intéressons à cette taxonomie car la nature de certaines préoccupations secondaires les rend vulnérables à se transformer en *préoccupations transverses* comme nous le verrons à la section 2.3.1. C'est cette catégorie qui posera des problèmes d'ordre architectural.

Préoccupations transverses

Les préoccupations peuvent généralement être séparées et encapsulées en suivant diverses techniques ou pratiques architecturales comme des *design patterns [GHJV94]*, des *styles architecturaux* (modèle en couches,N-tiers, MVC, etc.), etc.

Cependant, avec les langages OO purs, il existe un certain nombre de préoccupations qui ne peuvent être totalement séparées et qui continuent à cohabiter avec les préoccupations principales des classes.

En fait, même en portant une attention particulière à la séparation stricte des préoccupations dans des méthodes et classes distinctes, il subsistera toujours, avec la programmation objet, des préoccupations indissociables et envahissantes.

Ces *préoccupations* sont alors qualifiées de *transverses*[2] ou *cross-cutting concerns* car elles coupent plusieurs modules sans pour autant être regroupées.

Problèmes et effets liés aux préoccupations transverses De par leur nature, les préoccupations transverses induisent généralement un *enchevêtrement* des préoccupations ainsi qu'un *éparpillement*. Cette pollution est particulièrement néfaste car elle affecte généralement négativement à la fois le couplage, la cohésion et induit une perte d'encapsulation.

- *L'éparpillement* (scattering) est le phénomène par lequel une préoccupation se trouve répartie dans plusieurs modules plutôt que d'être dûment regroupée et encapsulée. Cet éparpillement contamine alors potentiellement des modules n'ayant, logiquement et en apparence, aucun lien avec cette préoccupation.

  Le premier problème majeur engendré par ce phénomène est la difficulté inhérente à circonscrire les modifications. En effet, puisque le code est éparpillé, il sera nécessaire de modifier plusieurs modules, même non liés logiquement, si la préoccupation devait changer. Plusieurs des modules seraient alors impactés indirectement augmentant la

2. Certains auteurs utilisent le terme préoccupations transversales en français.

complexité de la modification ainsi que le risque d'erreur. Ainsi, un développeur désirant effectuer une modification, même simple en apparence, risque de ne pas prévoir l'ensemble des modules qui seront affectés.

Le deuxième problème est une duplication presque inéluctable du code. En effet, le plus souvent, le code éparpillé dans les différents modules est généralement du code identique ou très similaire. Le plus souvent, il s'agit du code nécessaire pour appeler la partie de la préoccupation qu'il a été possible de regrouper.

- *L'enchevêtrement* (tangling) va souvent de pair avec *l'éparpillement* (scattering) du code. En effet, puisqu'il n'est pas possible de bien regrouper la préoccupation, les différents morceaux éparpillés se trouvent alors forcément mélangés avec du code répondant à d'autres préoccupations.

  Ainsi, il n'est pas rare d'avoir au sein d'une même classe ou d'une méthode des considérations comme la journalisation, la vérification des accès, la synchronisation, etc. en plus du code requis pour effectuer le travail nécessaire à la préoccupation principale.

  Ce mélange de préoccupations entre directement en contradiction avec le SRP (section 2.3.1) et nuit donc significativement au principe *d'encapsulation*.

  De plus, *l'enchevêtrement* a tendance à coupler inutilement la classe avec des entités qui n'ont pas de lien direct avec la préoccupation principale. Cette dépendance n'étant pas directement liée à la préoccupation principale, elle risque de ne pas être intuitive pour un développeur tiers.

La combinaison de ces deux facteurs a des répercussions importantes sur la maintenabilité du logiciel :

- Augmentation du couplage. De plus, comme nous l'avons vu, ce couplage est souvent produit par les dépendances entre modules qui ne sont pas logiquement liées.
- Diminution de la compréhensibilité et de la clarté du code. Le code étant contaminé, il est forcément plus long et il est plus difficile de repérer les lignes contribuant réellement à la fonctionnalité que l'on cherche à comprendre. Les préoccupations transverses se trouvent donc à polluer également visuellement le code.
- Dans le même ordre d'idées, les modifications sont également plus complexes car elles requièrent une compréhension de l'ensemble des préoccupations présentes dans un module lorsqu'une modification doit être effectuée. Et ce, même s'il s'agit d'une modification à une préoccupation principale simple qui, normalement, devrait avoir une portée limitée.

  Même avant de procéder à l'altération du code, il sera plus difficile de précisément identifier les modules qui seront impactés. Ce dernier problème est directement lié à la difficulté de respecter le SRP.
- Etc.

Un point intéressant à noter est que ces deux effets contribuent mutuellement à augmenter les problèmes de couplage. Ainsi, les préoccupations transverses devenant de plus en plus

nombreuses et imbriquées, l'impact d'une simple modification peut avoir un effet boule de neige sur un grand nombre de modules.

Ces deux facteurs combinés nuisent considérablement à la maintenabilité du logiciel. Nous pouvons alors supposer que dans la plupart des cas, plus la taille et la complexité du logiciel augmentent, plus les effets peuvent être dévastateurs.

Exemple 2

Dans le cas de la journalisation, par exemple, il sera nécessaire d'appeler la méthode « log » à chaque fois que l'on désire journaliser une information (*éparpillement*).

De plus, ces appels seront forcément localisés avec du code répondant à une autre préoccupation principale (*enchevêtrement*).

Supposons une application dans laquelle tenter d'obtenir la fiche d'un utilisateur inexistant doit être journalisée. La méthode `trouverUtilisateur` pourrait ressembler à ceci :

```
public Utilisateur trouverUtilisateur(id) {                                    1
  utilisateur = trouverUtilisateurDansMemoire(id);                             2
  if(utilisateur == null)                                                      3
    logger.warn(String.format("Utilisateur %s inexistant", id));               4
}                                                                              5
```

L'appel vers le « `logger` » ne contribue pas à la préoccupation principale de la classe et encore moins de la méthode. Le code se retrouve ici *contaminé* par la préoccupation *« journalisation »*. On peut clairement constater qu'il y a *enchevêtrement* des préoccupations et ce, fort probablement, à plusieurs endroits dans la classe.

Remarquons que la classe a maintenant une dépendance vers la classe `Logger`, augmentant ainsi le couplage. Cette dépendance n'est pourtant pas nécessaire pour trouver un utilisateur, qui s'avère être la préoccupation principale, et donc la seule raison qui motive l'existence de la méthode.

Cet exemple très simple, voire simpliste, nous permet d'entrevoir les conséquences des préoccupations transverses dans un système réel, complexe et impliquant des centaines de classes, en plusieurs niveaux, voire distribuées.

2.3.2 Orientation aspect (AO)

L'orientation aspect est une réponse à ces problèmes liés aux préoccupations transverses. Il est désormais possible de séparer ces préoccupations et de les regrouper en modules et ainsi obtenir les avantages architecturaux mentionnés ci-haut. On appelle alors ces modules des « aspects ».

Différentes technologies et langages de programmation sont disponibles pour répondre à ce nouveau paradigme. Les techniques varient mais cela implique très souvent d'inverser la dépendance et de permettre à l'aspect de décider lui-même où s'exécuter plutôt que de laisser les autres modules lui faire appel explicitement. Ainsi, les autres modules n'ont aucune (ou peu selon la technologie) connaissance de l'aspect. Cela n'est pas forcément le cas pour tous les langages aspects mais c'est le mécanisme privilégié par la majorité des solutions les plus populaires.

En ce sens, il s'agit d'une sorte de classe à laquelle on ajoute la possibilité de décider où doivent s'exécuter ses méthodes.

2.3.3 AspectJ

AspectJ est un langage et un tisseur (compilateur AO qui tisse les aspects aux endroits où l'aspect doit s'injecter) très populaire et probablement l'un des plus complets.

Étant donné que notre objectif est de déterminer le maximum d'impacts potentiels sur un projet, nous utiliserons cette technologie pour notre étude. Il s'agira également de la seule implémentation disponible pour notre cadre applicatif, bien que la conception permette le support éventuel de d'autres tisseurs.

2.3.4 Adoption dans l'industrie

Malgré plusieurs avantages indéniables de l'AOP, cette technologie reste toujours relativement peu adoptée dans l'industrie comparativement à l'OO. Et ce, malgré que l'AOP soit, en quelque sorte, une amélioration par rapport à l'OO.

Du point de vue strictement de la modélisation OO, les développeurs n'auraient, théoriquement, que peu à perdre à adopter l'AOP. En effet, l'AOP offre des moyens supplémentaires pour d'atteindre les idéaux de l'OO afin de mieux ségréguer et modulariser les responsabilités et les rôles.

Quelques études démontrent les effets bénéfiques de l'AOP sur la qualité des logiciels, et particulièrement, sur la maintenabilité (voir chapitre 3). Néanmoins, plusieurs hypothèses ont été avancées concernant l'intérêt mitigé des développeurs à migrer à l'AOP. La complexité est un facteur souvent invoqué et probable. On peut également facilement émettre comme hypothèse l'inclusion moins répandue et omniprésente dans les cursus académiques.

Cependant, peu d'études ont été réalisées afin de vérifier les difficultés rencontrées par les développeurs lorsque ces derniers désirent utiliser l'AOP dans des conditions réelles. Quelques succès de migration dans d'importantes entreprises ont été présentés.

Cet aspect est exploré en détail dans l'étude que nous avons menée et qui est décrite au chapitre 3

2.4 Agile

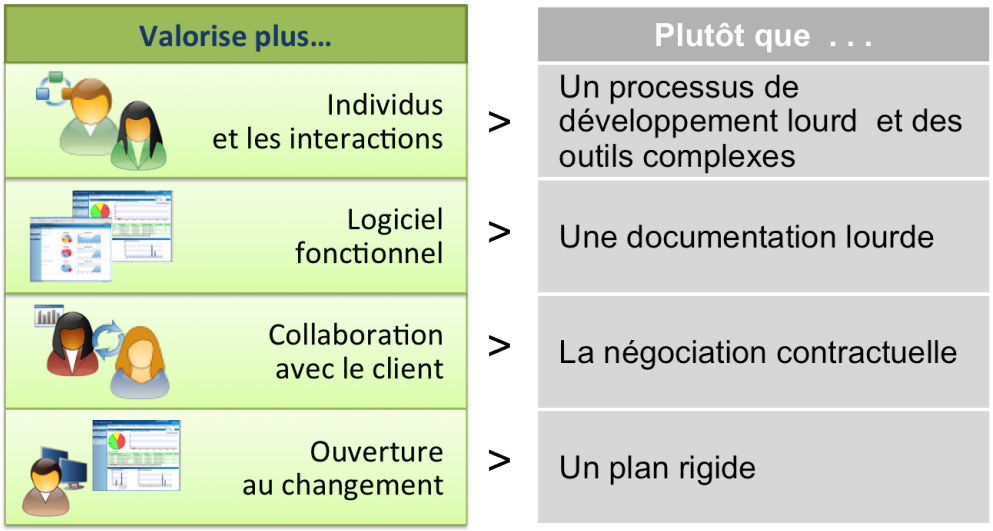
Telles qu'introduites à la section 1.2, les méthodes Agiles forment une famille de processus logiciels répondant à une série de valeurs et principes (voir section 2.4.1).

Il existe différents processus Agile répondant à ces valeurs. Ces processus peuvent également parfois se combiner car ils s'appliquent à des niveaux différents : organisationnel, gestion, équipe, pratiques de développement. Ainsi, plusieurs entreprises combinent Lean, Scrum et certaines pratiques de XP.

L'objectif de ce mémoire n'est pas de se positionner face aux bienfaits ou non de l'Agilité. Nous exposerons donc très rapidement les valeurs et certaines caractéristiques requises afin d'établir le contexte nécessaire pour la bonne compréhension de notre projet de recherche.

2.4.1 Valeurs

En 2001, 17 personnes ont signé un manifeste [BBvB+01] établissant les quatre valeurs de l'Agilité. Les valeurs proposées sont les suivantes :



Notons qu'il est important de lire les lignes en considérant l'importance relative de l'affirmation de gauche par rapport à celle de droite. Ainsi, bien que les signataires accordent de l'importance aux éléments de droite, ils privilégient davantage les éléments de gauche.

2.4.2 Acceptation du changement

L'acceptation du changement est l'un des fondements du développement Agile. En développement logiciel, on reconnaît généralement que le coût des modifications augmente avec le temps. Dans les approches traditionnelles, on tente de minimiser ce problème en effectuant une analyse poussée en début de projet afin de réduire les risques de modifications majeures plus tard dans le projet. Ce principe est cependant basé sur la prémisse qu'il est possible de réaliser une telle analyse juste et correspondant aux besoin du client.

Or, les Agilistes voient deux problèmes avec cette vision [3] :

- Il n'est souvent pas possible de traduire parfaitement le besoin d'un client car les besoins menant à une solution informatique moderne sont complexes et abstraits. Il s'agit de besoins d'affaires, de compétitivité ou de connectivité extrêmement complexes. Ces besoins sont donc très difficiles à définir car ils sont reliés à la structure, culture et vision de l'entreprise. Ainsi, prévoir les besoins TI revient un peu à prévoir l'avenir de l'entreprise et de ses différents intervenants. Un logiciel est une solution abstraite qui permet de répondre à des besoins abstraits. Les Agilistes veulent profiter de cet avantage pour offrir des solutions flexibles.

- Les besoins informatiques sont généralement implicitement changeants car ce sont des problèmes abstraits et reliés au domaine des affaires qui est le plus souvent volatile. Par exemple, un pont est une solution d'ingénierie au besoin de traverser une rivière. Or, la réalité physique ne changera pas pendant les cinq prochaines années. Ainsi, même si le processus d'analyse est long, le besoin ne changera probablement que très peu. De plus, une fois construit, il est peu probable qu'il soit nécessaire d'ajouter des fonctionnalités demandant de changer radicalement son « architecture ».

  Ainsi, même s'il est possible de capturer parfaitement le besoin, ce dernier risque de changer avec le temps, ce qui demandera quand même inévitablement de modifier la solution. La question, selon eux, n'est donc pas de savoir comment « empêcher » le changement puisqu'il est inévitable mais plutôt comment l'accepter et réduire le coût des changements.

Cependant, l'hypothèse Agile ne peut être vraie que si des pratiques de développement sont mises en place afin de maintenir un coût du changement faible tout au long du projet comme le montre la figure 2.1.

3. Cette vision est également appuyée sur la production rapide de valeur (retour sur investissement) qui permet de générer des revenus à partir d'une dépense plus rapidement puisque la réalisation a débuté beaucoup plus tôt. Cependant, l'objectif de ce mémoire n'est pas d'expliquer l'argumentaire en faveur de l'Agilité. Nous présentons cette vision uniquement afin de comprendre le contexte d'une équipe Agile et des liens avec les pratiques de développement.

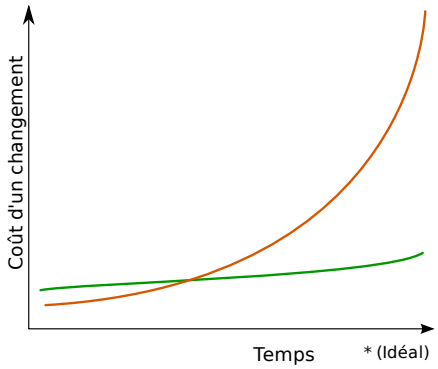


FIGURE 2.1: Coût des changements dans le temps (schématisé)

Note 2

En fait, cette vision, basée sur les autres domaines du génie, assume que le coût de l'analyse et de la conception d'un plan est moins onéreux que la phase de réalisation. Une erreur dans le plan d'un pont découverte après sa construction, par exemple, coûtera extrêmement cher à réparer. Or, selon les Agilistes, la nature du logiciel (maléabilité, non-destructivité, abstraction, etc.) rend, dans les faits, la réalisation moins coûteuse que l'analyse. Modifier l'architecture d'un système ou ajouter une fonctionnalité oubliée coûte beaucoup moins cher. Par contre, faire l'analyse d'une fonctionnalité logicielle qui n'est pas utilisée est un gaspillage qui coûte très cher.

2.4.3 Pratiques de développement Agiles

Telles qu'introduites à la section précédente, les méthodes Agiles reposent sur le principe que les équipes mettent en place les pratiques et techniques nécessaires afin d'assurer un coût des changements faible tout au long du projet.

Cela implique une série de pratiques recommandables [4] :

- Maintenir le code propre (Clean Code [Mar08a]) ;
- Favoriser un design simple [5] ;
- Respects des principes architecturaux (SOLID Principles [Mar02a], etc.) ;
- *Réusinage (refactoring) constant ;*
- *Tests automatisés* (incluant le TDD et l'ATDD [6]) ;
- Programmation en paires ;
- Etc.

Presque toutes ces pratiques permettent directement ou indirectement de maintenir le coût d'un changement bas tout au long d'un projet. Plusieurs de ces pratiques sont, par contre, interconnectées et ne peuvent être sélectionnées à la pièce. Ainsi, il sera presque impossible de faire d'importants réusinages sans avoir de très bons tests automatisés car il sera difficile de savoir si le réusinage a brisé quelque chose.

Notons que plusieurs de ces pratiques sont directement liées avec la maintenabilité du logiciel. L'objectif étant de produire, itérativement et incrémentalement, un logiciel répondant à des critères élevés de maintenabilité et d'évolutivité.

C'est pour cette raison que la première partie de nos recherches consiste à déterminer si l'AOP peut interférer avec ces pratiques fondamentales au succès d'une équipe Agile à long terme (voir chapitre 3). De plus, la deuxième partie de nos recherches vise à créer un outil permettant aux développeurs de tester unitairement (en isolation) leurs aspects afin qu'il leur soit possible d'appliquer cette pratique malgré la présence d'aspects (voir chapitres 4 et 5).

2.5 Tests dans un projet Agile

Les tests logiciels permettent de s'assurer qu'un logiciel fonctionne correctement. En fait, les tests peuvent servir à répondre à deux grandes questions :

- Est-ce que le logiciel est correctement construit (interne) ?
- Est-ce le bon logiciel (réponse aux besoins) ?

4. Il n'existe pas de liste officielle mais ceci est une liste des pratiques les plus recommandées dans l'industrie et la littérature. Il s'agit d'un échantillon et ces pratiques ne sont pas toujours adoptées ensemble.

5. Il existe des règles de design simple. Notons que simple n'implique pas simpliste ou facile...

6. L'emploi de l'ATDD et/ou du BDD est cependant apparu comme pratique assez récemment dans la communauté.

Afin de répondre à ces questions, divers types de tests (système, charge, intégration, unitaire, acceptation, ...). Chacun de ces types sont complémentaires et visent à tester le logiciel sous différents angles. Dans le cadre de cette recherche, nous nous concentrerons sur les tests unitaires tel qu'expliqué à la section 2.5.2.

Étant donné le fait que les tests sont non destructifs avec des logiciels, contrairement à la majorité des autres domaines du génie, il est possible de les exécuter *très fréquemment* et *directement sur le produit livrable.* Malheureusement, les tests sont très souvent sous-employés dans l'industrie, possiblement pour diverses raisons historiques, de la formation des informaticiens mais également à cause de la culture de cette industrie.

Pourtant, le taux de succès et la qualité des projets informatiques sont désastreux comme le montre certaines études effectuées par le Standish Group [Sta09] et d'autres [Sco10]. Bien que les données exactes de ces rapports soient critiquables, il est assez clair que la qualité n'est pas toujours au rendez-vous (bogues, ...) ou peut générer de nombreux dépassements de coûts [Sal12, Les11].

Afin de tenter d'améliorer la qualité, mais aussi afin de restaurer la confiance envers l'industrie, les processus Agile ont mis, avec les années, de plus en plus d'emphase sur les tests. En plus de les utiliser abondamment, la communauté Agile a beaucoup contribué à perfectionner et inventer de nouvelles techniques, outils et pratiques de tests dont le TDD [7], le BDD [8], JUnit, etc.

Note 3

Sans entrer dans les détails, notons que la communauté Agile recommande de maximiser [9] la qualité des tests automatisés. Ceux-ci présentent, notamment, l'avantage d'être reproductibles et ainsi de permettre des livraisons fréquentes (concept à la base de l'Agilité). Pour cette raison et considérant que ce projet de recherche vise les tests unitaires, nous assumerons généralement qu'un test est, par définition, automatisé pour la suite de ce mémoire.

Les sections subséquentes, présenteront certains types, techniques et pratiques de tests pertinents dans le contexte de ce projet de recherche tels qu'ils sont *généralement utilisés et préconisés dans un projet Agile.*

7. Test Driven Development
8. Behavior Driven Development

2.5.1 Taxonomie des tests

Il existe plusieurs types et catégories de tests logiciels. On peut ainsi classer les tests par leur cible, la personne intéressée par ceux-ci, leur visibilité, etc. Notons que ces taxonomies ne sont pas normalisées à proprement dit et il existe un certain flou sur la définition en fonction de l'entreprise, du milieu et même de l'interlocuteur.

De plus, ces catégories ne sont pas exclusives puisqu'il s'agit de différentes manières de catégoriser. Ainsi, un test peut se classer dans plusieurs catégories.

Finalement, il est important de mentionner que ces catégories sont généralement complémentaires. Ainsi, faire uniquement des tests unitaires apportera des bénéfices mais ne permet pas de tester toutes les facettes du logiciel. Une approche d'essais devrait comprendre des tests de différentes catégories pour répondre à différentes questions ou risques.

Pour la suite de ce mémoire, nous allons tenter de coller le plus possible sur les définitions et les façons de faire d'un projet Agile et d'employer les techniques répandues au sein de la communauté en raison de l'objectif de recherche défini en introduction.

2.5.2 Tests unitaires

On définit généralement les tests unitaires comme un essai permettant de tester une unité en isolation. La définition d'une unité peut varier mais en programmation objet, on la définit généralement comme étant une classe. En d'autres mots, l'objectif du test unitaire est de tester la classe sous test (CUT [10]) en l'isolant du reste du système.

L'objectif principal des tests unitaires est de s'assurer que chaque petit morceau du logiciel (une unité) fonctionne indépendamment des autres. Cela permet de trouver des erreurs éventuelles rapidement et de les corriger à la source. En effet, une anomalie découverte par un test de haut niveau demandera plus d'effort de déverminage et de recherche afin de trouver l'emplacement et la cause du problème.

Avoir des tests unitaires isolés, permet de s'assurer d'une fondation solide afin de bâtir le système avec confiance. Peut-on être certain que cette classe fait ce qu'elle doit faire ? Puis-je réutiliser sans crainte cette classe ? De plus, un test unitaire apporte également une forme de documentation exécutable.

Avoir uniquement des tests de haut niveau reviendrait à placer une caméra devant un édifice pour détecter un éventuel incendie. Alors que dans ce cas, un détecteur de fumée placé dans chaque appartement permet de détecter un feu dans une unité plus rapidement avant sa propagation.

10. CUT : Classe Under Test

Voici un résumé des caractéristiques d'un test unitaire :

- Pour : les développeurs
- Par : les développeurs
- Quoi : une unité isolée
- Quand (en Agile) : pendant le développement
- Couverture recherchée (en Agile) : haute (typiquement 80-95%)
- Pourquoi :
  - augmenter la confiance dans chacun des « blocs » (classe) composant le logiciel ;
  - faciliter le réusinage ;
  - faire ressortir les problèmes tôt ;
  - communiquer.

Notons que le niveau d'isolation recommandé dépend fortement du contexte mais aussi des auteurs. Le principe étant que plus l'isolation est forte, moins le test devra être touché lors de modifications au logiciel. Idéalement, un test unitaire ne devrait être touché uniquement que si l'unité testée doit être modifiée.

Isolation

Isoler la classe testée nous permet de s'assurer, dans un premier temps, qu'il n'y a pas de problème au sein même de la classe en question. Cela offre un certain nombre d'avantages :

- Permet d'avoir une suite de tests unitaires extrêmement rapides à exécuter car il n'y a pas d'appel au réseau, à la base de données, etc.
- Le test est autonome et ne requiert pas une configuration complexe pour l'exécuter. Par le fait même, les tests peuvent être sans états et il n'est donc pas nécessaire de remettre constamment les valeurs par défaut ou considérer un état initial.
- Permet de se concentrer sur les comportements de la classe même et d'éliminer les autres sources d'erreurs potentielles.
- Favorise une bonne conception simple et découplée car une classe trop couplée sera difficile à tester. Un développeur aura donc naturellement tendance à privilégier un couplage faible.
- Dans le cas d'une anomalie détectée dans un test intégré, cela implique que le problème se situe fort probablement dans la collaboration entre les objets et non dans le comportement même des objets. Cela réduit considérablement le temps nécessaire pour trouver les problèmes.

Tout cela, dans un contexte de projet Agile, permet d'obtenir une conception plus simple et souple qui supportera plus facilement le changement et le réusinage. Cette souplesse est

également favorisée par le fait que le développeur peut rapidement écrire un test en même temps que l'écriture de la classe testée. Sans cela, il est souvent nécessaire de mettre en place un environnement et de coder la couche d'infrastructure avant d'écrire un premier test pouvant couvrir toutes les lignes modifiées ou ajoutées [11].

Procédure

Pour tester une unité (une classe) en isolation, on procède comme suit :

- le test T pilote la CUT en appelant des méthodes sur celle-ci ;
- on vérifie ensuite si le comportement résultant est celui attendu.

Le comportement résultant peut être interne et/ou externe. C'est-à-dire qu'il peut modifier des états internes de la classe testée (CUT), retourner ou encore déclencher des interactions avec d'autres objets (vers les dépendances) comme le montre la figure 2.2.

Dans ce dernier cas, nous pourrons vérifier ce comportement grâce aux Mocks (voir section 2.5.3).

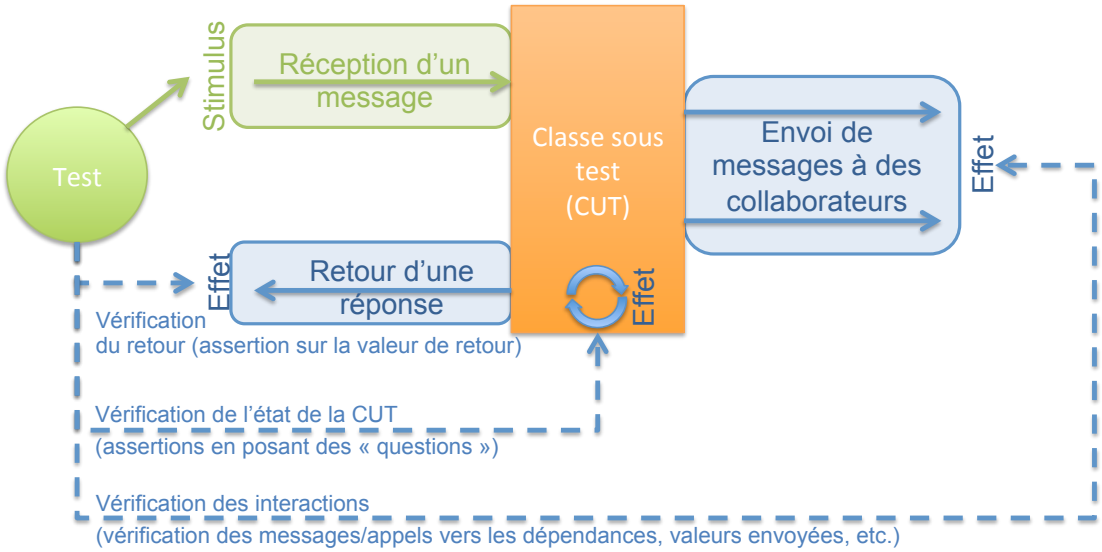


FIGURE 2.2: Vérifications lors d'un test unitaire.

2.5.3 Mocks

Nous avons vu précédemment à la section 2.5.2 qu'un test unitaire devrait être réalisé en isolation. Cela implique qu'une classe doit être « déconnectée » de ses dépendances afin de ne tester que la classe visée et non ce qui est utilisé (délégation) par celle-ci.

11. L'objectif de ce mémoire n'est pas de discuter des avantages ou inconvénients des approches Agile, du TDD, etc. Nous exposons ici simplement et trop succinctement les avantages notés ou promis aux équipes Agile afin de leur permettre d'accepter le changement et répondre aux valeurs de l'Agilité.

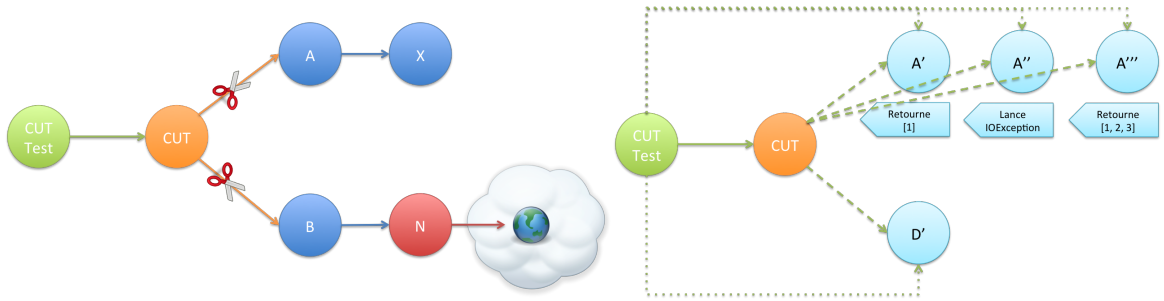


FIGURE 2.3: Fonctionnement d'un mock.

À première vue, couper une classe de ses dépendances semble presque impossible car, après tout, elle ne peut pas fonctionner, voire compiler, sans celles-ci. L'astuce consiste donc à remplacer les dépendances par des faux équivalents. En d'autres termes, nous utiliserons une dépendance du même type mais qui agit à titre de simulateur pour la durée du test comme le montre la figure 2.3.

Concrètement, dans un langage objet, le polymorphisme permet cela car il nous est possible de faire utiliser par la classe testée (CUT) n'importe quelle sous-classe de ses dépendances [12].

L'autre question est de savoir par quoi remplacer les dépendances. L'objectif général est de les remplacer par des « faux » qui possèdent les mêmes interfaces (même comportements offerts) que leur dépendance réelle. Une fois appelé, un « faux » peut se comporter de différentes façons [Fow07] :

- ne rien faire ou retourner « null » (stub) ;
- simuler une implémentation réduite de la dépendances (fake) ;
- répondre une réponse pré-programmée (mock).

Bien qu'un « stub » soit parfois suffisant, il est souvent nécessaire et souhaitable d'utiliser un « mock ». L'idée est de pré-programmer le « mock » de manière à ce qu'il se comporte tel que voulu afin de créer le cas de test désiré chez l'appelant (la CUT). On programmera ainsi différemment le « faux mock » pour les différents cas de tests désirés.

Notons que la classe « mock » est du type d'une dépendance et non pas de la classe testée. Une erreur courante des débutants est de créer un « mock » de la classe à tester. Dans ce cas, le test se trouve à tester un « faux » ! C'est plutôt la classe testée qui doit *utiliser* les mocks comme dépendance.

En plus de piloter les cas de tests, le mock permet également de vérifier l'effet de la classe testée (CUT) sur ses dépendances. En effet, nous avons déjà expliqué à la section 2.5.2 qu'un

12. À quelques exceptions près en fonction du langage et du respect du OCP (Open-Closed Principle) par le développeur. Il existe alors des méthodes de contournement mais l'objectif n'est pas d'exposer les fins détails de la technique du « mocking » dans ce mémoire.

test unitaire doit vérifier l'effet à la fois sur l'état interne de la CUT mais également ses effets externes sur les dépendances (voir la figure 2.2).

Étant donné que le mock simule une dépendance, les méthodes du mock seront appelées au moment où la réelle dépendance l'aurait été. L'idée est donc de programmer le mock de manière à noter les appels (ou autres interactions) reçus. Ainsi, le test pourra demander au mock s'il a bien reçu les appels escomptés avec les bons paramètres, etc.

Exemple 3

Supposons la classe sous test (CUT) suivante :

```
class EventReservationService {
  // ...
  public void reserveTicketFor(member):
    try {
      Seat seat = room.reserveFirstSeatAvailable();
      member.assignSeat(seat);
    } catch(NoSeatLeftException e) {
      waitlist.add(member);
    }
  }
}
```

Lors d'une demande de réservation, deux cas sont possibles :

- au moins une place est toujours disponible ;
- aucune place n'est disponible.

Pour accomplir sa tâche, la classe utilise deux dépendances externes :

- une salle (Room) ;
- une liste d'attente (Waitlist).

Remarquons ici que le cas (place disponible ou non) dépend de la réponse de la dépendance « Room ». De plus, les effets de la méthode sont externes et, par conséquent, le test devra vérifier les actions posées sur les dépendances afin de vérifier si le comportement espéré est obtenu. Notons que sans mocks et isolation, le test impliquerait d'aller vérifier l'effet global résultant de la chaîne de dépendances ce qui le rendrait complexe et fortement couplé.

Voici un test (partiel) possible afin de vérifier ces cas :

```
class EventReservationServiceTest {
  private Member aMemberMock;
  private Seat seatAvailable;
  private Room roomMock;
  private Waitlist waitlistMock;
  private EventReservationService service;

  @Before public void setup() {
    aMemberMock = mock(Member.class);
    seatAvailable = new Seat();
```

```
    roomMock = mock(Room.class);
    waitlistMock = mock(Waitlist.class);

    service = new EventReservationService(roomMock, waitlistMock);
  }

  @Test public void enoughSeatWhenReserveShouldAssignFirstSeatAvailable {
    // Configure le mock pour retourner seatAvailable quand la méthode
    // reserveFirstSeatAvailable est appelée. Le mock enregistrera l'appel
    // mais ne fera rien d'autre que de retourner cette valeur.
    willReturn(seatAvailable).given(roomMock).reserveFirstSeatAvailable();

    // Assumons que service a été configuré pour utiliser les Mocks...
    service.reserveTicketFor(aMemberMocked);

    // Ceci est vrai uniquement si assignSeat a été appelé
    // 1 fois avec seatAvailable comme paramètre.
    verify(aMemberMock).assignSeat(seatAvailable);
  }

  @Test public void noSeatWhenReserveSouldAddToTheWaitList() {
    willThrow(new NoSeatLeftException()).given(roomMock).reserveFirstSeatAvailable();

    service.reserveTicketFor(aMemberMocked);

    verify(waitlistMock).add(aMemberMocked);
  }
}
```

Remarquons dans cet exemple comme le mock « roomMock » a servi afin de recréer les deux cas possibles à tester (lancer une exception ou non). De leur côté, les mocks « waitlistMock » et « aMemberMock » ont servi à vérifier si le comportement espéré est obtenu.

L'une des difficultés avec les mocks consiste à trouver un moyen de les injecter dans la classe testée. En effet, la classe testée doit être configurée de manière à utiliser le mock plutôt que la vraie implémentation. L'injection est plus ou moins difficile en fonction de la manière dont la classe a été conçue. Dans l'exemple précédent, l'injection était simple car le constructeur de la classe acceptait ses dépendances en paramètres. Notons que la difficulté d'injecter un mock est étroitement liée à la qualité de la conception (design) et les difficultés devraient être considérées comme étant des indices (odeurs) potentiels d'une mauvaise conception [13].

13. Cela est lié au principe de design comme les S.O.L.I.D. Principles, l'injection de dépendances et les fondements mêmes de l'orientation objet. Cependant, ceci sort de la portée de ce mémoire.

Finalement, notons que bien qu'il soit possible de créer des mocks manuellement (en créant des vraies classes héritant de la dépendance), l'industrie utilise maintenant majoritairement des cadres applicatifs (frameworks) automatisant la création de *mocks virtuels*. Ces cadres créent alors des classes virtuelles dynamiquement (par programmation) en utilisant la réflexivité des langages modernes. *Notez que c'est précisément cette approche que nous emploierons dans le cadre de notre projet de recherche afin de permettre les tests unitaires d'aspects* (voir chapitre 5).

2.5.4 Tests d'acceptation

Un test d'acceptation est un test écrit dans le but de s'assurer qu'une fonctionnalité répond réellement aux besoins du client. En d'autres mots, il permet de répondre à la question : « Avons-nous développé la bonne fonctionnalité ».

Normalement, en mode Agile, il est rédigé dans un langage compréhensible par le client, voire écrit par lui-même si possible. Pour y arriver, il existe différentes techniques comme le BDD et des outils comme Cucumber qui ont fait école.

Peu importe l'outil employé, l'idée est d'exprimer le besoin utilisateur sous la forme d'exemples (des scénarios d'essais) écrits en langage naturel du domaine d'affaires (DSL). Un développeur écrit ensuite une couche de programmation que l'on nomme « Fixture ». Cette colle permet de lier le langage naturel avec l'application. Ainsi, l'exécution du scénario écrit sous la forme d'un exemple s'exécutera en pilotant l'application et vérifiant les comportements attendus.

Pour faciliter la traduction du langage naturel mais également afin de faciliter la lecture des scénarios par des humains, nous recommandons généralement de suivre une structure « Given-When-Then » [14]. Cette structure consiste à découper le test en trois grandes « actions » : état initial (étant donné), action à tester (quand) et comportement attendu (alors).

Exemple 4

Voici un exemple de scénario textuel rédigé sous la forme Given-When-Then :

```
ÉTANT DONNÉ un solde bancaire de 1000$
    ET aucune marge de crédit
    QUAND je fais un paiement de 1001$
    ALORS le paiement est refusé
```

L'exemple ci-haut montre bien comment cette technique permet à un utilisateur non programmeur de rédiger son besoin sous la forme d'exemples simples pouvant éventuellement être exécutés.

14. Cette structure est très employée par la communauté Agile suite à l'apparition du mouvement BDD et du Gherkin. Notons que cette structure ne s'applique pas uniquement aux tests d'acceptation mais également à d'autres types de tests, dont les tests unitaires.

Voici un résumé des caractéristiques d'un test d'acceptation :

- Pour : le client
- Par : le client (idéalement) ou avec sa collaboration
- Quoi : une fonctionnalité (une User Story en mode Agile)
- Couverture recherchée (en Agile) : toutes les fonctionnalités [15]
- Pourquoi :
  - s'assurer que le logiciel fait ce qu'il est supposé faire ;
  - s'assurer de répondre au besoin du client ;
  - s'assurer qu'aucune fonctionnalité n'a été brisée.

Tests d'acceptation dans le projet Mock4Aj

Le chapitre 5 présentera l'outil Mock4Aj développé dans le cadre des travaux de recherche présentés dans ce mémoire. Nous avons écrit des tests d'acceptation pour chacune des spécifications (User Stories) que nous avons implantées. Ainsi, le cadre applicatif (framework) est entièrement couvert par des tests qui s'assurent que le besoin est respecté.

Cela fournit également une très bonne documentation afin de comprendre ce que le cadre applicatif fait et comment l'utiliser. Nous avons porté un grand soin pour que ces tests soient faciles à lire et à comprendre. La simplicité des tests d'acceptation démontre d'ailleurs qu'il est relativement facile d'utiliser le cadre applicatif sans connaître les détails d'implantation de ce cadre complexe.

Notons que le tout est également testé à l'aide de tests unitaires en plus des tests d'acceptation. Tous ces tests sont disponibles et peuvent être consultés avec le code source [16]. Les tests d'acceptation sont regroupés par User Stories.

15. Note : Il ne s'agit pas d'une mesure des lignes couvertes. Toutes les fonctionnalités n'impliquent pas toutes les lignes de code puisque le test d'acceptation ne vise pas à tester tous les cas du point de vue du développeur.
16. Le code est disponible sous licence libre LPLG

2.5.5 Tests d'aspects

Plusieurs études [ABA04, Zha02, ZA07, MBB07] traitent des tests d'aspects. La complexité de tester des aspects vient, surtout, des « points de coupure » qui n'existent pas en orientation objet. Ce concept est assez nouveau car il inverse le sens de déclaration par rapport au flot d'appels réel. En d'autres mots, l'aspect décide lui-même où s'injecter et donc, où il y aura appel vers l'aspect.

Afin de tester adéquatement un aspect, il est important de vérifier que l'aspect :

- a le bon comportement (contenu des greffons) ;
- s'exécute au bon moment (point de coupure).

C'est ce dernier cas qui est particulièrement nouveau et pour lequel il faut trouver de nouvelles techniques et technologies.

Le chapitre 4 expliquera plus en détail les problèmes et pistes de solutions liés aux points de coupures.

Tests unitaires d'aspects

La majorité des études citées ci-haut proposent des solutions afin de tester les aspects mais rarement d'un point de vue unitaire. En effet malgré que les auteurs emploient le terme « unitaire », il ne s'agit pas d'un test unitaire du point de vue Agile. C'est-à-dire qu'ils ne testent pas l'*aspect en isolation*. La majorité de ces études utilisent, dans les faits, le terme unitaire dans le sens de *tests développeurs* dans la nomenclature agile.

Notons que cela n'enlève aucune valeur à ces tests qui sont, même en Agile, nécessaires. Seulement, les agilistes favorisent la construction de suites séparées avec des techniques de tests différentes en fonction du niveau et de l'objectif. Nous avons déjà discuté brièvement à la section 2.5.2 des raisons qui motivent ce choix.

N'oublions pas également que beaucoup d'agilistes pilotent la construction du code en écrivant d'abord un morceau de test (TDD et ATDD). Cela permet une alternance très rapide entre l'écriture d'un morceau de test (jusqu'à un échec), puis un morceau de code (quand le test passe) et finalement, une courte phase de réusinage. Ce processus, connu sur le nom de TDD [17], fait en sorte que la phase de test n'est plus placée après la réalisation d'une fonctionnalité. Plusieurs gains ont été rapportés avec cette pratique [Sin06] [18] et cette pratique est maintenant assez répandue .

De plus, en Agilité les diagrammes UML et la conception initiale sont toujours présents [Fow04b] mais servent plus à la communication qu'à la représentation détaillée d'un système. Les TDD

17. Test Driven Development
18. D'autres études et témoignages existent mais notre objectif n'est pas de statuer sur l'intérêt ou non de la pratique mais plutôt de nous assurer que notre approche ne nuit pas à ceux qui l'utilisent dans un projet Agile.

et cette vision excluent donc beaucoup d'approches basées sur le code existant ou sur les modélisations effectuées en amont. Les praticiens du TDD utiliseront plutôt les tests pour piloter et documenter la conception [FP09a].

L'article technique de N. Lesiecki [Les05] résume bien les difficultés liées aux tests unitaires d'aspects et certaines approches possibles. Il y décrit un certain nombre d'approches dont une technique utilisant les mocks. C'est, en gros, cette approche que nous adopterons pour la réalisation de notre cadre applicatif. Cette technique a également été supportée par des discussions au sein de la communauté d'AspectJ comme étant une solution à préconiser [Bod04a].

Le détail de cette technique est décrite exhaustivement à la section 4.2.1. Les problèmes liés aux tests unitaires d'aspects sont, quant à eux, décrits au chapitre 4.

2.6 Environnement de développement

2.6.1 Support au développement dans un projet Agile

De nos jours, les développeurs industriels utilisent des IDEs offrant de puissants outils de support au développement. Ces outils font désormais partie du quotidien des développeurs, contribuent à leur productivité et surtout permettent à ces derniers d'effectuer des tâches de maintenance en minimisant les risques de bris.

Parmi ces outils, nous retrouvons le support au réusinage qui permet, notamment de renommer des méthodes ou des classes, etc. L'IDE s'assurera de alors renommer toutes les invocations en utilisant l'arbre de compilation (pas uniquement textuelles). Des IDEs comme Eclipse peuvent même suivre la trace dans plusieurs projets dépendants, etc.

En plus des outils de réusinage, il faut également considérer l'auto-complétion, les outils pour les tests automatisés, etc. Ne pas avoir ces outils est un pas en arrière pour plusieurs développeurs et peut être un frein à l'adoption d'une technologie.

De plus, dans un cadre Agile, plusieurs de ces outils sont d'une importance capitale car ils permettent de simplifier l'évolution permanente du code à moindre coût. Ainsi, les développeurs n'ont presque rien à perdre à remanier le code, encore et encore, pour assurer le meilleur design et une clarté du code qui suit et évolue avec le temps et les modifications. Par exemple, les noms doivent changer afin de toujours représenter correctement les nouveaux comportements, etc. Cela permet également de ne pas avoir peur de diviser (extraire) des classes, ajouter des interfaces, etc. afin d'avoir le design qui correspond toujours le plus possible aux nouveaux besoins plutôt que de dupliquer ou ajouter des rustines (patches).

2.6.2 Impacts sur l'environnement de développement

La majorité des applications modernes en entreprise utilisent des cadres applicatifs ou mécanismes complexes. Ils font du Web, des clients riches, des systèmes distribués, des systèmes

organisés en services (web services) avec SOAP ou REST, etc.

Exemple 5

Voici un exemple de ce qu'une application offrant des services REST en Java pourrait utiliser. Ces technologies sont assez standards pour des applications d'entreprises et la plupart des projets d'une certaine envergure ont ce type de besoins.

- Un conteneur JEE (Java JEE Container)
- Spring (pour l'injection de dépendances)
- Hibernate avec support JPA2 (ORM pour la base de données)
- Jersey (JAX-RS pour REST)
- XStream (sérialisation XML)
- Etc.

En plus, les développeurs utilisent potentiellement plusieurs outils pour favoriser les tests, le développement et le déploiement :

- Maven (pour la compilation, dépendances (projets), déploiement, ...)
- JUnit (pour les tests)
- Jenkins (serveur d'intégration continue)
- Git (pour la gestion des sources)
- Eclipse (comme IDE avec le support du réusinage, etc.)

Par conséquent, l'introduction d'une nouvelle technologie ne doit pas entrer en conflit avec ces cadres applicatifs ou outils. Certains de ces outils sont assez poussés et utilisent des mécanismes complexes de Java ou encore prennent le contrôle du processus de complication (ex. : Maven et Jenkins).

Certains problèmes risquent donc d'arriver si la nouvelle technologie modifie des mécanismes clés de Java ou s'injecte dans le processus de compilation. C'est le cas, en AOP, de certains tisseurs d'aspects comme AspectJ car ils remplacent le chargeur de classes (Class Loader en tissage au chargement, LTW) ou le compilateur (en tissage à la compilation, CTW). On peut constater également des perturbations avec des analyseurs de code (métrique de couverture des tests, etc.) et des intégrateurs continus comme Jenkins ou Hudson.

Ce type de perturbation peut briser le processus de développement de l'équipe et nuire à ses pratiques de développement. Nous tenterons d'évaluer ces perturbations sur les pratiques Agile dans l'étude présentée au chapitre 3.

Chapitre 3

Évaluation des besoins

Ce chapitre résume les résultats d'une étude que nous avons menée en 2009 afin d'avoir une idée des irritants découlant de l'utilisation de l'AOP au sein d'une équipe Agile. Il s'agit d'une étude exploratoire dont les résultats ont été publiés [BL10] en 2012.

3.1 Introduction

Dans les dernières années, l'AOP a profité d'une certaine adoption en industrie mais souvent plus comme solution d'appoint pour contourner des limitations ou de mauvais designs [HK02, SLB02, SB05] que pour améliorer l'architecture en éliminant les préoccupations transverses.

En dehors de cette timide adoption, la littérature montre plusieurs cas où l'AO a été utilisée avec succès [HJ09, KM99]. Plusieurs recherches ont démontré [BH08, BCP+05, MS07] les bénéfices de l'AO sur le design et la qualité des logiciels. Cependant, ces avantages ne semblent pas se concrétiser par une large adoption dans l'industrie.

Plusieurs raisons peuvent expliquer ce phénomène telles que la complexité induite et la courbe d'apprentissage nécessaire. Étant donné que la majorité des développeurs industriels n'ont pas appris l'AOP lors de leur formation, il est important d'essayer de déterminer quels sont les obstacle rencontrés par ces derniers quand ils veulent utiliser l'AOP pour la première fois.

Certains de ces facteurs ne sont peut-être pas liés à la technologie même et peuvent même provenir de perceptions. Et c'est pourquoi nous avons monté une étude exploratoire afin d'accumuler quelques pistes pour avoir une meilleure idée des aspects pouvant être améliorés. *Ainsi, nous pourrons poursuivre nos travaux en essayant de cibler un problème potentiel et ainsi augmenter nos chances de fournir une contribution significative offrant des bénéfices quotidiens et immédiats dans un milieu industriel.*

Nous avons donc voulu mettre en place une première étude exploratoire qui s'appliquerait sur un projet d'une certaine taille et utilisant un processus logiciel Agile (voir chapitre 2). Nous nous intéressons particulièrement à cet aspect car, malheureusement, beaucoup d'études

réalisées ne précisent pas le processus utilisé par les équipes ou encore sont trop courtes ou se déroulent dans un environnement trop contrôlé pour que le processus puisse réellement influer [BH08, MS07, Mul05].

Pourtant, l'utilisation de l'AOP peut avoir différents impacts selon le processus utilisé. Par exemple, puisque l'Agilité s'appuie fortement sur le réusinage constant et les tests (voir chapitre 2), il devient important de savoir si l'utilisation de l'AOP favorise ou freine ces piliers du développement Agile.

Déjà, nous pouvons émettre comme hypothèse que l'AOP peut aider pour tester des comportements difficiles ou encore pour injecter des Mocks. Par contre, nous pouvons nous attendre à ce que le réusinage des aspects avec leurs points de coupures (pointcuts) soit plus difficile que le réusinage du code Java brut considérant les outils actuels.

3.2 Travaux reliés

Certaines études analysent déjà l'impact de l'AO sur la qualité, le design ou la gestion des projets [BH08, BCP+05, DPC01]. Cependant, la majorité de ces études se limitent à certaines métriques spécifiques et le contexte de l'étude se limite généralement au minimum pour évaluer ces métriques. Ces recherches sont très importantes pour évaluer les effets exacts mais ne fournissent pas une bonne idée des impacts dans le contexte d'un vrai projet industriel (incluant même les aspects humains, culturels et la réalité de gestion des entreprises).

Il existe plusieurs recherches concernant le réusinage et les tests en AO [KGRX08, KS04, WHH08, XZ06, ZA07, MBB07]. Notons que très peu [KGRX08, KS04] rendent leur outil disponible et fonctionnel pour qu'une équipe industrielle l'installe et l'utilise aisément quotidiennement.

Peu d'études ont été réalisées pour évaluer les effets de l'AO sur un processus Agile. Il y a quelques années, Madaysky et Szala [MS07] ont publié une expérimentation sur les impacts de l'AOP sur l'efficacité et la qualité du design en utilisant l'eXtreme Programming. Par contre, le contexte de l'expérimentation était très contrôlé et ne représente pas vraiment une équipe industrielle réelle puisqu'ils n'y a que trois participants à l'étude et un seul d'entre eux utilisait l'AOP.

Une autre étude conduite par J. Muller et al. [Mul05] évaluait l'impact de l'AOP sur les itérations. Cependant, l'étude a été conduite sur des itérations très courtes pour environ 71 minutes de projet. L'ampleur est également très limitée avec seulement 434 lignes de codes écrites pour l'AOP. De plus, le même développeur devait réaliser le même projet en AOP et en OOP. On peut donc penser que la différence de temps observée pourrait provenir de l'habitude acquise par le développeur.

En comparaison, notre étude est conduite sur 26 développeurs sur une période de 3 mois (13 semaines). Nous avons également recréé un contexte d'équipes pour se rapprocher un peu plus de la réalité en industrie.

Plus récemment, quelques rapports ont été publiés sur les applications industrielles de l'AOP [HJ09, KM99, PCG+08]. Un article récent [HJ09] décrit comment Siemens a utilisé l'AOP, les bénéfices ainsi que les problèmes qu'ils ont rencontrés.

Notre expérimentation est similaire mais se déroule dans un environnement un peu plus contrôlé et surtout plus représentatif de débutants en AOP, ce qui est notre objectif. Nous avons également plusieurs équipes différentes qui travaillent à réaliser le même projets (mêmes besoins). Cela nous permet de mieux identifier ce qui est commun. De plus, notre étude est réalisée dans un contexte Agile.

3.3 Contexte de l'étude

L'étude a été réalisée dans le cadre d'un cours d'Architecture logicielle de niveau de premier cycle universitaire à l'hiver 2009. Les données ont été extraites d'une évaluation formative d'un projet conduit dans le cours. Le projet durait toute la session et était obligatoire avec un très fort pourcentage de la note sur ce dernier. Le questionnaire a été complété à la dernière semaine de la session par les étudiants.

3.3.1 Le cours

Le cours Architecture logicielle est un cours de premier cycle qui traite de divers sujets entourant l'architecture : design patterns, styles architecturaux, AOP, etc.

Les cours préalables à ce cours sont des cours de programmation et d'analyse en orientation objet (OO) qui traitent d'UML, OO et de processus. Plusieurs étudiants ont également suivi un cours de qualité logicielle portant sur la qualité du code, de l'architecture ainsi que sur les tests. Par conséquent, la grande majorité des équipes avaient au moins une personne familière avec les tests unitaires, les tests d'acceptation, les mocks, etc.

Les caractéristiques principales du cours étaient :

- 3 heures de cours par semaine ;
- 2 séances (en début de session) portaient sur l'AOP (total de 6h) ;
- des laboratoires traitaient d'AOP (2h) ;
- une formation a été donnée sur les processus Agile et Scrum par un formateur certifié (3h) ;
- des revues de projets à chaque deux (2) semaines où les étudiants devaient démontrer un produit fonctionnel et discuter de la planification de la prochaine itération (sprint)

avec le client (les enseignants). Les enseignants donnaient également des commentaires formatifs sur le projet, le code et l'architecture.

Les séances de cours et laboratoires portant sur l'AOP traitaient des motivations, applications, bonnes et mauvaises pratiques, etc. En plus de cela, le langage AspectJ a été présenté avec des exercices. La présentation d'AspectJ incluait des concepts avancés (cflow, percflow, etc.).

3.3.2 Le projet de session

Le projet de la session d'hiver 2009 consistait à réaliser un portail financier web. Ce portail devait permettre de gérer un portefeuille de titres boursiers. L'utilisateur pouvait entrer le nombre de titres possédés ou encore simplement surveiller des titres ou indices financiers.

Pour réaliser le projet, il était nécessaire de créer une application web utilisant plusieurs sources financières en temps réel (ex. : Google Finance, Yahoo Finance, etc.).

Un exemple d'interface graphique (UI) tiré du projet d'une équipe est donné à la figure 3.1). L'interface devait inclure, entre autres, un tableau de bord qui permettait à l'utilisateur de gérer son portfolio et de suivre en temps réel la valeur des titres suivis. Il pouvait ajouter et retirer des titres et modifier la quantité d'actions qu'il possède. Il était également possible de suivre des titres (actions et indices boursiers). Finalement, les informations financières devaient être mises à jour automatiquement en arrière plan (sans intervention du côté serveur).

Le projet complet était composé de 32 histoires (User Stories). Chaque histoire était associée avec des critères d'acceptation tels que c'est généralement fait en Scrum. Chaque équipe pouvait collaborer (négocier) avec les propriétaires du produit (Product Owner – PO) afin d'établir leur planification de la prochaine itération (sprint) de deux (2) semaines en considérant la priorité (PO) et l'estimation (équipe).

Fidèle à une planification Agile, la quantité de fonctionnalités n'était pas fixée à l'avance puisque cette quantité varie avec la vélocité des équipes. Comme prévu, aucune équipe n'a réalisé les 32 histoires du carnet du produit (product backlog). De plus, les équipes étaient libres de faire leur propre choix technologiques et architecturaux.

Finalement, il est arrivé durant la session que le propriétaire du produit change d'idée concernant certaines histoires ou en change la priorité afin de recréer un contexte réaliste d'un projet industriel Agile. Les étudiants étaient avertis de cette possibilité et il leur avait été conseillé d'adopter un processus qui incluait et valorisait les réusinages continuels et le TDD (Tests Driven Development). Ils étaient également encouragés à écrire des tests d'acceptation en plus des tests unitaires.

3.3.3 Les participants

À la session d'hiver 2009, 30 étudiants étaient initialement inscrits et 2 ont abandonné le cours pendant la session. Le tableau 3.1 et la figure 3.2 montrent la composition du groupe inscrit.

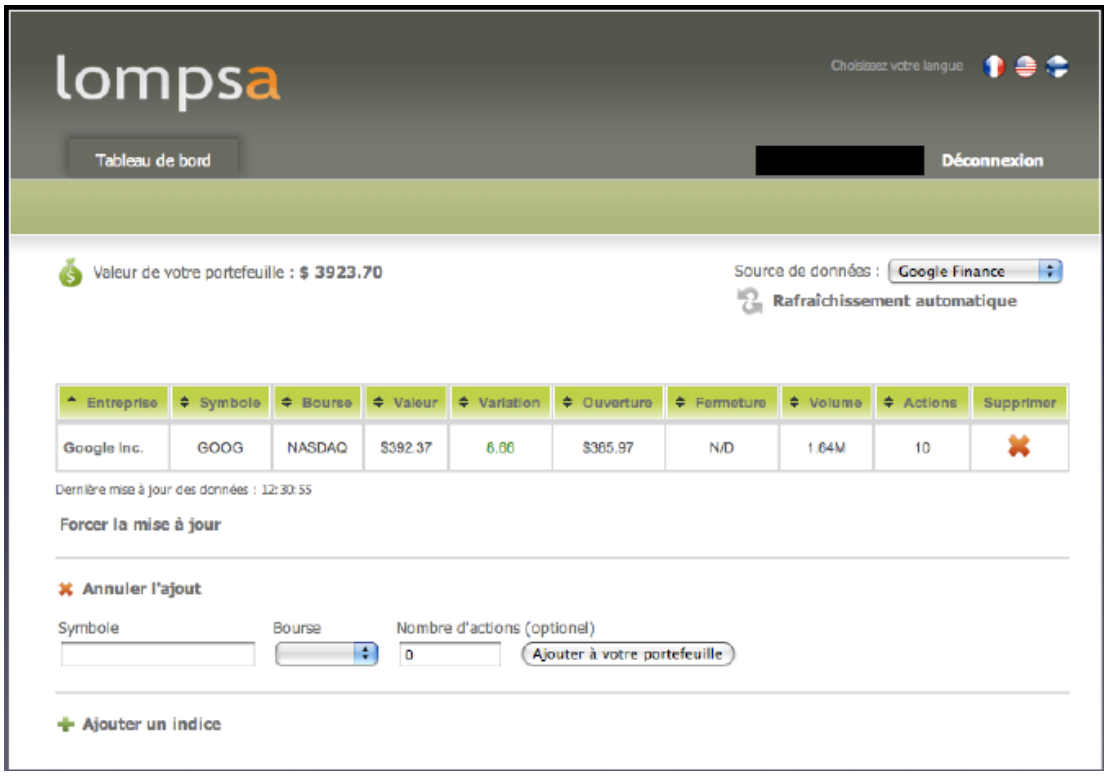


FIGURE 3.1: Interface graphique du projet de session (équipe 1).

Il est à noter que les étudiants ayant abandonné le cours ont été retirés.

Programme d'étude	Nombre	%
Bacc. en génie-logiciel	16	57.14
Bacc. en informatique	6	21.43
Échange international	5	17.86
Certificat en informatique	1	3.57
Total	28	100

TABLE 3.1: Composition du groupe

Selon le cheminement normal, les étudiants de génie logiciel ont déjà réalisé des projets académiques sur une session complète en grand groupe avec obligation de résultats.

Le cours est obligatoire pour les étudiants en génie logiciel (GLO) et il est prévu dans le cheminement à la 3e année du baccalauréat de 4 ans. Les étudiants des autres programmes, principalement sciences informatiques (IFT), peuvent prendre le cours en option. Ces derniers prennent généralement le cours à la dernière année d'un baccalauréat de 3 ans.

Voici les principales distinctions significatives selon le programme d'étude :

– Expérience Java : la plupart des étudiants en GLO ont fait des projets en Java contrairement aux étudiants en IFT et internationaux qui ont principalement travaillé en C/C++.

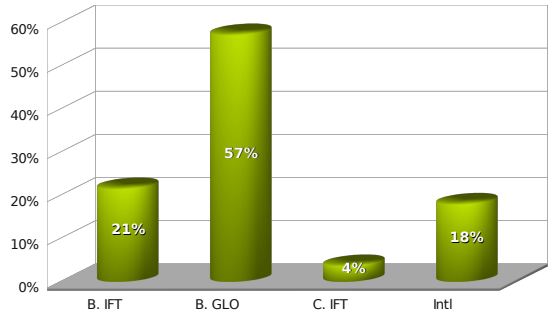


FIGURE 3.2: Composition du groupe par programme d'étude

– Expérience en OO : les programmes de GLO et IFT ne partagent pas les mêmes cours d'analyse et conception OO. La version GLO est donnée par le département de génie informatique et inclut un projet de session substantiel en Java. La version IFT se concentre plus sur l'UML et des exercices pratiques de 2 ou 3 semaines.

– Étudiant à temps plein : nous avons observé que plusieurs étudiants en IFT travaillent à temps partiel en même temps que leurs études. Comparativement avec les étudiants de GLO et internationaux qui tendent majoritairement à être à temps plein comme étudiant.

3.3.4 Processus Agile

Les équipes suivaient un processus Agile [1] autant que possible en suivant les recommandations des enseignants. Les pratiques enseignées étaient tirées d'eXtreme Programming (XP) et Scrum. Ces deux méthodologies sont compatibles en ce sens que XP focalise sur les pratiques d'ingénierie alors que Scrum est centré sur la gestion de projet et d'équipe. De plus, le processus était légèrement modifié afin de l'adapter à un contexte académique.

Évidemment, il n'est pas possible d'être entièrement certain que les équipes ont adopté toutes les pratiques car certaines comme le TDD ou les réunions quotidiennes ne sont pas clairement visibles de l'extérieur de l'équipe. Par contre, les pratiques impliquant le code ou encore la planification des itérations ont été évaluées.

3.3.5 Technologies

Le projet était réalisé en Java, Eclipse et AspectJ (AJDT). N'importe quel cadre applicatif web libre (Open Source web framework) pouvait être utilisé par les équipes. Toutes les équipes ont utilisé Google Web Toolkit (GWT) sauf une qui a choisi Spring.

1. En fait, en début de session, ils avaient le choix entre Agile ou RUP. Toutes les équipes ont voté en majorité pour adopter un processus Agile. Cela nous permettait d'avoir une meilleure chance d'adoption par les équipes en raison de leur implication dans le choix.

Le produit devait fonctionner à la fois sous GNU/Linux et Microsoft Windows autant pour le serveur que pour le client (navigateur).

À l'exception de Java, Eclipse et AspectJ, chaque équipe était entièrement libre d'utiliser les technologies qu'elle désirait. Les enseignants ont toutefois fait des recommandations à des équipes et la petite variété peut s'expliquer par l'adoption d'outils utilisés dans les cours précédents. Voici en résumé les technologies les plus utilisées :

- toutes les équipes ont utilisé JUnit pour les tests unitaires ;
- toutes les équipes ont utilisé Concordion pour les tests d'acceptation ;
- une seule équipe a utilisé Maven ;
- toutes les équipes ayant correctement utilisé des Mocks ont utilisé Mockito.

3.3.6 AOP

L'utilisation de l'AOP était obligatoire dans le projet mais, dans le but d'améliorer le design et minimiser les préoccupations transverses. Les étudiants n'étaient pas contraints d'utiliser l'AOP pour des usages particuliers mais ils avaient l'obligation de produire un bon design logiciel en minimisant le couplage, en augmentant la cohésion et en respectant les principes d'isolation [Mar02b]. Pour arriver à ces fins, ils pouvaient procéder comme bon leur semblait en utilisant des patrons (design patterns), l'AOP, etc. Ces critères d'évaluation étaient connus par les étudiants dès le début de la session.

Les cours et laboratoires sur l'AOP ont eu lieu en début de session (semaine 2 à 4) pour qu'ils puissent acquérir les habiletés nécessaires dès le début du projet.

Exemples d'utilisation de l'AOP

Voici les usages les plus courants de l'AOP faits par les équipes :

- injection de Mocks pour les tests ;
- gestion des exceptions ;
- gestion des erreurs de communication ;
- validation des entrées ;
- surveillance (monitoring) des sources de données financières ;
- simulation de comportement pour les tests (changement du flot, lancement d'exceptions, etc.) ;
- gestion de la concurrence (threads).

3.3.7 Formation des équipes

Les équipes étaient composées de 4 à 5 étudiants inscrits au cours. La formation a été réalisée en suivant les étapes suivantes :

1. Chaque étudiant devait remplir un formulaire à propos de son niveau de connaissance et d'expérience avec différents concepts et technologies ;

2. L'étudiant pouvait choisir ses coéquipiers et devait indiquer s'il avait déjà travaillé ou non dans le passé avec chacun d'eux ;

3. Il n'était pas nécessaire de former une équipe complète mais ils étaient conscients qu'ils avaient plus de chance d'être séparés s'ils ne formaient pas une équipe complète.

Après une période d'une semaine pour soumettre leur équipe, les enseignant ont procédé à la formation finale en fonction des propositions reçues. La formation était basée sur les critères suivants :

1. Donner priorité aux équipes complètes (5 personnes).

2. Donner priorité aux sous-groupes ayant déjà travaillé ensemble. La motivation était de ne pas séparer des étudiants ayant déjà eu une expérience positive de travail en commun et qui souhaitaient retravailler ensemble.

3. Considérer une répartition et un niveau viable des connaissances au sein d'une équipe en se basant sur les réponses individuelles aux questions de l'auto-évaluation. L'objectif était de ne pas mettre des projets à risque en raison du manque de connaissances. Par exemple, nous nous sommes assurés que toutes les équipes avaient au moins une personne ayant déjà fait du Java.

Au terme du processus, la majorité des équipes sont restées intactes et la grande majorité des préférences ont été respectées.

Étant donné ce processus, la distribution des compétences dans les équipes n'était évidemment pas uniforme car nous avons observé que les étudiants ayant un profil académique similaire et plus d'expérience ont tendance à se regrouper. Cela permet cependant, d'une certaine façon, de reproduire la réalité entre entreprises qui sélectionnent en fonction de profils similaires.

La figure 3.3 montre la composition des 6 équipes en fonction du programme d'étude des étudiants. Il est fort probable que le programme d'étude joue un rôle important dans le type de compétences et dans la maturité des étudiants.

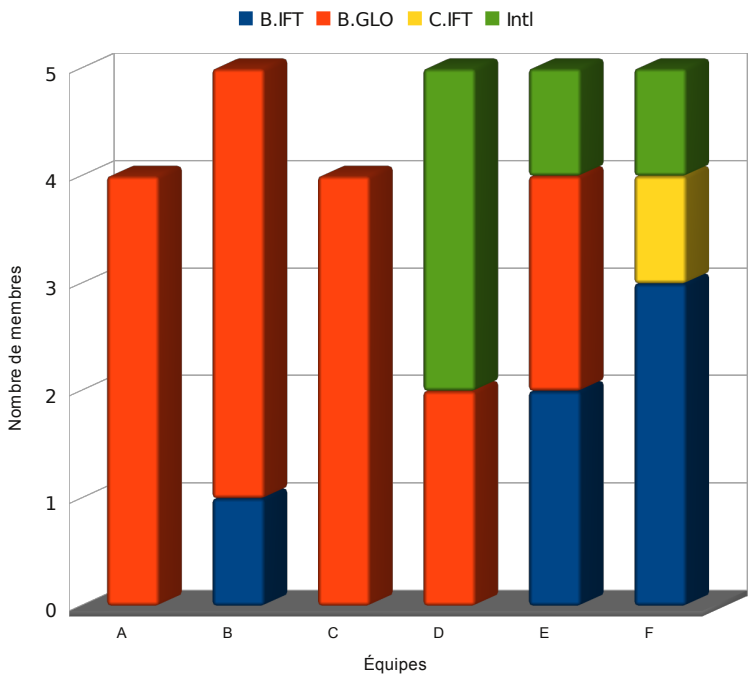


FIGURE 3.3: Composition des équipes par programme d'étude

3.3.8 Évaluations

La durée du projet était de 13 semaines. Chaque itération était fixée à deux (2) semaines sauf pour la dernière (3 semaines). Après chaque itération, les équipes faisaient une revue d'itération ainsi qu'une planification pour la suivante. Ces rencontres se déroulaient comme suit :

1. L'équipe fait une démonstration de leur produit fonctionnel.
2. L'équipe discute d'une planification pour la prochaine itération. Le propriétaire du produit (les enseignants), discute avec l'équipe et s'entend sur l'engagement ;
3. Les enseignants regardent le code et donnent des commentaires sur l'architecture, la qualité du code ou la gestion du projet.

Pour être préparés, les enseignants prélevaient le code source quelques jours avant la rencontre afin de le regarder. Les enseignants pouvaient ainsi critiquer, donner leur option et poser des questions pour aider les équipes. Cependant, les enseignants ne donnaient jamais de solutions complètes aux équipes mais uniquement des conseils afin de les aider.

Le contenu de cette rencontre n'était pas évalué (formatif) afin de favoriser le dialogue sous une forme de coaching. Seule la présence et la participation était évaluée pour ces rencontres.

Deux livraisons étaient prévues pendant la session : l'une au milieu et l'autre à la fin. Aucun document n'était demandé. Seul le code prélevé était évalué par les enseignants qui scrutaient chaque ligne pour évaluer : le style, la qualité du code, les anomalies, la lisibilité, la simplicité,

le bon fonctionnement, etc. Quelques métriques étaient également utilisées comme le couplage, la cohésion, le graph de dépendances, etc. Le respect par l'équipe des principes OO (SRP, Loi de Demeter, OCP, etc.) était également évalué.

À la fin de la session, les étudiants devaient présenter le résultat de leur travail à leurs collègues et aux enseignants.

3.4 Analyse des résultats

À la fin de la session, chaque étudiant devait compléter un formulaire d'évaluation (formatif) et donner des commentaires sur le projet de session. Les données de cette étude sont tirées de ce formulaire anonyme.

Le formulaire a été complété lors de la dernière semaine de la session, soit environ une semaine avant la fin du projet. Le questionnaire était anonyme et les données d'identification qui auraient pu être ajoutées par erreur ont été retirées avant l'évaluation des données.

3.4.1 Contenu

Le formulaire avait deux objectifs dans la réalité. Le premier était d'obtenir des rétroactions et commentaires sur le projet afin de permettre aux enseignants d'améliorer la formule pour l'année suivante. Le deuxième était d'obtenir des informations concernant l'adoption, la pertinence et le niveau de compréhension concernant l'AOP, le processus Agile et les technologies. Plusieurs questions étaient dissimulées afin d'obtenir des informations sur l'AOP dans un processus Agile.

La question comptait 3 pages dont 4 sections. Chaque section comptait entre 4 et 12 questions avec certains tableaux à compléter.

3.4.2 Étude

Au moment de compléter le formulaire, les étudiants n'étaient pas au courant que ce dernier pourrait servir dans le cadre d'une étude sur l'AOP et les méthodes Agiles.

Lors de l'analyse des résultats, seulement les questions concernant l'étude (AOP et Agile) ont été considérées et les questions concernant l'amélioration du cours ont été retirées. De plus, l'analyse a été effectuée sur des données complètement rendues anonymes.

3.4.3 Analyse statistique

La méthodologie d'analyse est la suivante :

- Par défaut, les pourcentages sont basés sur le nombre de répondants excluant les absents. Ce faisant, nous évitons de faire des hypothèses sur le profil des absents ;

– Certaines questions demandaient d'évaluer le degré de facilité pour réaliser certaines tâches dans une situation donnée. L'échelle était alors de 1 (difficile) à 5 (facile). Pour des fins d'analyse, les données ont généralement été groupées comme suit :

1-2 : Relativement difficile

3 : Neutre

4-5 : Relativement facile

– Les pourcentages ont été arrondis pour exclure les décimales ;

3.5 Résultats

3.5.1 AOP

Caractéristique de l'échantillon

Il y a eu 26 questionnaires de complétés dont les caractéristiques sont les suivantes :

– 89% des répondants ne connaissaient rien à l'AOP avant le début de la session alors que les 12% restants n'avaient qu'une connaissance théorique. Aucun n'avait utilisé de l'AOP dans un projet.

– À la fin de la session, les étudiants auto-évaluent leur niveau de compréhension de l'AOP comme suit :

Débutant : 15%

Intermédiaire : 35%

Bon : 50%

Expert : 0%

– Sur un plan individuel, seulement 12% n'ont jamais écrit de code AspectJ durant la session dans leur projet. Dans les 88% qui ont écrit du code AspectJ, 39% n'ont écrit que quelques morceaux.

– Pour les équipes, une (1) seule équipe n'a pas livré de code AspectJ fonctionnel. En fait, ils l'ont essayé et du code AspectJ a été inclus mais ce dernier ne fonctionne pas correctement.

Satisfaction et pertinence

La vaste majorité des répondants considèrent que l'AO a été utile et pertinent en utilisant AspectJ comme le montre ces résultats :

– 92% recommanderaient à un collègue d'apprendre l'AOP ;

– 100% seraient tentés d'utiliser l'AO dans un futur projet, mais 62% le feraient avec prudence ;

- 96% réutiliseraient l'AO et AspectJ si le même projet était à refaire ;
- 71% pensent que l'AO est un paradigme prometteur mais qui requiert d'être perfectionné ;
- D'un autre côté, 95% considèrent que l'AO est suffisamment mature pour être enseigné dans un cours de premier cycle universitaire ;
- 100% considèrent l'AO comme un paradigme et une technologie offrant de multiples possibilités.

Apprentissage et utilisation

La facilité d'apprentissage a été évaluée comme suit :

- La majorité des répondants (74%) estiment qu'il n'était pas plus difficile d'apprendre l'AOP que l'OO ou encore que Java ;
- D'un autre côté, même après avoir appris l'AOP et appliqué le concept dans leur projet, 67% pensent que l'AO est complexe ;
- Après avoir eu du matériel et activités pédagogiques (diapositives, laboratoires, références sur Internet, etc.), les répondants sont divisés concernant le manque ou non de documentation disponible concernant l'AO et AspectJ.

  En fait, 58% considèrent qu'il manque de documentation, contre 42% qui l'estiment suffisante.

AspectJ

La grande majorité des répondants considèrent toujours qu'écrire un aspect avec AspectJ est plus difficile que d'écrire une classe « standard » en Java. Mais plusieurs considèrent que cela est causé par leur manque d'expérience comparativement à l'OO et non pas par la technologie elle-même :

- 25% estiment qu'il est aussi facile de développer un aspect en AspectJ qu'une classe Java ;
- Dans les 75% restants qui considèrent plus difficile d'écrire un aspect, 89% indiquent que c'est exclusivement causé par un manque d'expérience.

Concernant la syntaxe d'AspectJ, 63% la considèrent facile à apprendre.

Finalement, la majorité (58%) estiment qu'il était relativement facile d'écrire un point de coupure (pointcut) qui apparie (match) les points de jonctions voulus.

3.5.2 Tests

Test de code Java conseillé

Concernant les tests du code Java conseillé par des aspects, 52% des répondants trouvent qu'il est relativement facile d'écrire des tests pour ce code contre 19% qui trouvent cela relativement difficile. Les 29% restants sont neutres.

Remarquons cependant que le taux d'abstention (19%) est assez élevé pour cette question et peut laisser croire que plusieurs étudiants ayant rencontré ce type de situation n'avaient pas l'expérience nécessaire pour répondre.

Tests d'aspects

Tous les répondants (100%) estiment qu'il est difficile de tester des aspects avec les technologies qu'ils utilisent (AspectJ, JUnit, Mockito). D'un autre côté, on ne sait pas si cela est causé par un manque de documentation.

Cela semble être un obstacle majeur pour un débutant avec l'AOP. Cela est particulièrement intéressant dans un contexte Agile où les tests sont une pierre angulaire de la méthode et qui permet de réduire la dette technique.

Utiliser des aspects pour tester

Concernant le fait d'utiliser AspectJ pour réaliser des tests de code Java :

- 67% considèrent que les tests ont été simplifiés par l'emploi d'AspectJ quand cela était approprié (ex. : injection de mocks). Il est important de remarquer le fort taux d'abstention sur cette question (42%) ce qui laisse croire que plusieurs n'ont pas fait suffisamment de tests sous ces conditions pour se prononcer ;
- D'un autre côté, seulement 35% trouvent qu'il est relativement facile d'écrire des tests aidés par des aspects. Dans les faits, le groupe est assez divisé sur cette question :

  Relativement facile : 35%

  Neutre : 35%

  Relativement difficile : 30%

- Sur une question plus spécifique, 52% considèrent qu'il est relativement facile d'injecter des mocks (avec Mockito et JUnit) à l'aide d'un aspect contre 17% qui trouvent cela difficile.

3.5.3 Architecture et qualité

Sur les impacts architecturaux de l'AOP, les résultats sont les suivants :

- 64% considèrent qu'utiliser l'AOP (AspectJ) augmente la clarté du code ;

- 61% considèrent qu'utiliser l'AOP (AspectJ) aide à augmenter la maintenabilité du logiciel contre 39% qui ne voient aucune amélioration. Le taux d'abstention étant assez élevé (31%), il faut considérer cette donnée avec prudence, surtout que le projet n'a pas été maintenu sur une longue période.
- 85% considèrent de manière générale que l'AOP améliore l'architecture d'un logiciel.

3.5.4 Réusinage

Réusinage d'aspects existants

Les répondants sont divisés concernant la difficulté de réusiner des aspects en cours de développement :

- 38% évaluent ce type de réusinage comme relativement facile, 29% relativement difficile et 29% sont neutres ;
- Le taux d'abstention est cependant un peu élevé (19%) ce qui laisse croire que plusieurs n'ont pas tenté ce type de réusinage.

Réusinage du code Java conseillé

- 19% trouvent qu'il est relativement facile de réusiner du code Java conseillé par un ou des aspects contre 29% qui trouvent cela relativement difficile et 52% qui sont neutres.
- Encore une fois, le taux d'abstention est encore élevé (19%) potentiellement pour les mêmes raisons.

3.5.5 Agile

Caractéristiques de l'échantillon

Le questionnaire indique que 72% des répondants savaient déjà ce qu'était l'Agilité avant de débuter le cours. Par contre, dans ces 72%, la vaste majorité n'avait jamais appliqué l'Agilité dans un projet à la lumière de ce qu'ils savent maintenant sur l'Agilité :

Jamais : 56%

Essayé : 22%

Partiellement : 17%

Sérieusement : 6%

À la fin de la session, la majorité des répondants évaluaient leur niveau de compréhension de l'Agilité comme étant relativement bon :

Débutant : 4%

Intermédiaire : 28%

Relativement bon : 68%

Expert : 0%

Satisfaction concernant l'Agilité

À la fin du projet, la satisfaction concernant les processus Agiles était de 100%. En fait, 100% des répondants seraient tentés d'utiliser un processus Agile dans le futur et seulement 12% le feraient avec des réserves ou avec prudence.

De plus, si le même projet était à refaire, 92% des répondants utiliseraient un processus Agile à nouveau. Les commentaires laissés portent à croire que les 8% restants ne croient pas que le processus s'applique bien dans un contexte académique car ils ne peuvent se dédier à temps plein au projet.

3.6 Discussion

Cette étude rapporte les opinions, impressions, difficultés et succès rencontrés par des étudiants qui sont de nouveaux utilisateurs de l'AO dans un contexte Agile.

La grande majorité des répondants semble être enthousiaste face à l'AOP et n'hésiteraient pas à l'utiliser dans de futurs projets. Ils considèrent également l'AOP comme étant un sujet que l'on devrait aborder dans un cursus de premier cycle universitaire.

Apprendre l'AOP et AspectJ semble être moins difficile que nous le pensions initialement. En effet, la plupart des répondants considèrent l'AOP comme étant complexe mais pas beaucoup plus difficile à apprendre que l'OOP ou Java. De plus, la majorité pense que les difficultés qu'ils ont eu proviennent plutôt du manque d'expérience. Finalement, la vaste majorité pense que l'AOP amène des bénéfices sur l'architecture du logiciel.

Tous (100%) les répondants semblent avoir eu des problèmes pour tester les aspects écrits en AspectJ avec les cadres applicatifs actuels. Par contre, la majorité semble penser que l'AOP peut aider grandement à tester du code Java. Malheureusement, le taux d'abstention est assez élevé sur cette question, probablement causé par le fait que plusieurs personnes n'ont pas eu à expérimenter ce cas.

Ces résultats concernant les tests sont particulièrement pertinents dans le cadre d'une méthodologie Agile car il s'agit de pratiques au coeur de l'Agilité et valorisée par une vaste majorité de la communauté. En effet, ces pratiques doivent être réalisées de manière routinière par les développeurs et, par conséquent, peut sérieusement être un frein pour une équipe pratiquant l'Agilité.

Ces équipes risquent donc d'être moins enclines à adopter l'AOP si elles ne sont pas capables de tester efficacement les aspects. Puisqu'il s'agit d'une bonne pratique bien ancrée dans les équipes Agiles que d'effectuer de bons tests unitaires, celles-ci risquent de préférer ne pas utiliser d'aspects ou d'en faire une utilisation limitée.

Il en va de même pour les pratiques de réusinage du code AspectJ qui peuvent également être une source de frustration pour une équipe Agile qui pratique cette discipline sur une base régulière.

Cela nous laisse croire qu'améliorer les outils de tests pour les aspects pourrait être un facteur important afin d'aider les équipes Agiles à embrasser l'AOP et plus précisément AspectJ. Comme les développeurs Agiles font beaucoup de tests unitaires (voir chapitre 2), nous pensons que de proposer un cadre applicatif qui pourrait répondre aux besoins d'un tel projet serait probablement un facteur permettant une meilleure adoption tout en minimisant les compromis nécessaires.

Nous tenons à noter au passage qu'il est intéressant de voir que les étudiants semblent avoir apprécié l'utilisation des méthodes Agiles dans un cadre académique de même que l'apprentissage de l'AOP.

Concernant cette étude, il serait intéressant de conduire une étude similaire auprès d'experts oeuvrant déjà depuis longtemps dans des équipes Agiles. Cela donnerait un autre aperçu en comparaison aux effets sur les développeurs qui ont peu été en contact avec l'AOP et l'Agilité. De plus, cela nous permettrait de voir si notre solution peut tenir la route auprès de personnes étant déjà habituées à travailler avec certains outils de tests.

Chapitre 4

Tests unitaires d'aspects à l'aide de Mocks

4.1 Motivations et objectifs

4.1.1 Problème initial

Notre étude (voir chapitre 3 et [BL10]) révèle une difficulté à tester des aspects avec les technologies industrielles courantes.

Contexte industriel et automatisation. Plusieurs recherches ont été menées en ce sens (voir 2.5.5 et 2.5.5) mais le fruit de ces efforts en est souvent resté au stade de prototypes ; empêchant ou rendant ainsi difficile leur utilisation par un développeur professionnel dans un cadre industriel où l'outil sera utilisé de manière extensive et marié avec d'autres technologies. Dans un contexte Agile, par exemple, un tel outil devra souvent fonctionner de pair avec des intégrateurs continus, des systèmes de déploiement automatisés et des cadres applicatifs (frameworks) de tests. Ce type d'outils ne peut être aisément remplacé sans induire d'importants coûts ; incluant la formation et la baisse de productivité, certes temporaire, mais inhérente.

Parmi ces cadres, il existait, entre autres, aUnit [the] et AJTE [YSM+05, ajt05]. Ces solutions n'ont malheureusement jamais été largement adoptées par la communauté et ne sont plus maintenues. Ces deux solutions avaient également un défaut important : elles requéraient une étape semi-manuelle afin de générer des artefacts. Bien entendu, cette étape est relativement simple et peut être scriptée. Cependant dans un contexte industriel cela peut s'avérer un problème car le processus de compilation, de test et de déploiement est normalement automatisé et intégré avec des outils comme Maven. Leur intégration requiert donc des efforts ou, pire, avoir des effets de bords importants une fois combinés.

Antinomie avec les modèles mentaux actuels. Un autre problème vient du fossé séparant souvent les méthodes de tests unitaires d'aspects par rapport aux techniques OO déjà maî-

trisées, employées et reconnues. Parfois, cette distance est inévitable car inhérente au concept même de l'orientation aspect. Cependant, il ne s'agit parfois que d'une barrière psychologique causée par l'ontologie (vocabulaire) choisie. Dans certains autres cas, c'est le fonctionnement, soit la manière d'écrire les tests (la technique), qui ne colle pas aux façons de faire et de penser usuelles, forçant ainsi les équipes à revoir leurs modèles mentaux afin de s'adapter.

Cela peut sembler n'être qu'un irritant passager mais devient un réel obstacle lorsque ces deux visions doivent être conjuguées au sein d'un projet, voire d'un même test. Cela complexifie les tests, les rendant ainsi moins intelligibles et plus difficiles à faire évoluer. En effet, même pour un développeur ayant une bonne maîtrise des modèles, leur combinaison augmente la confusion puisqu'il doit constamment basculer d'un modèle mental à un autre.

Productivité et adoption. Avec le temps, les équipes développent des habitudes et deviennent de plus en plus performantes avec les techniques qu'elles maîtrisent. En effet, l'utilisation récurrente des techniques et technologies rend les développeurs accoutumés à celles-ci et leur permet de s'élever au-dessus de la technique afin de se concentrer sur le problème à résoudre. Par conséquent, l'introduction de technologies dont le fonctionnement diffère grandement de ce qui est déjà en place demande un effort cognitif qui risque de diminuer momentanément la productivité. Les ingénieurs devront inclure cette perte de productivité ainsi que la complexité indue dans leur évaluation des gains et coûts associés à l'introduction de l'AO et de son cadre de tests.

Notons que l'accoutumance peut parfois être un problème dans une équipe Agile car elle peut freiner l'innovation. Cependant, le fait d'utiliser toujours la même méthode de rédaction des tests au sein d'un même projet permet également l'uniformité et la constance entre les tests d'un même projet. Cela présente des avantages en permettant, dans une certaine mesure, d'augmenter la compréhensibilité et la productivité, en plus de permettre d'éviter des erreurs [Mar08a].

Propositions et demandes de la communauté. En 2004, Andrian Colyer a proposé à la communauté [Col04] de développeurs un cadre applicatif se rapprochant des techniques connues et utilisées en OO. Il y expose les requis et critères nécessaires, selon lui, pour tester unitairement (en isolation) un aspect. S'en suivra un essai appelé aUnit [the] mais qui ne sera jamais réellement terminé. De plus, plusieurs des nombreux commentaires émis par la communauté n'ont pas été considérés.

À l'époque, certains proposaient déjà de réutiliser le concept de Mocks adapté aux aspects [Bod04b]. Ils proposaient une syntaxe [Nic04] se rapprochant de celle employée par certains cadres applicatifs pour les Mocks. Malheureusement ces propositions n'ont pas été retenues dans le prototype initial de aUnit [the].

4.1.2 Hypothèses

Tel que discuté aux sections 2.5.2 et 2.5.3, un test unitaire devrait normalement tester le module ciblé isolément des autres. Typiquement, un projet Agile OO, utilisera des Mocks Objects (objets factices) [MFC01] afin d'y parvenir.

Par conséquent, un projet possédant des suites de tests unitaires respectant les règles de l'art [1] [MFC01, FP09b, CP02] devrait normalement déjà employer des cadres applicatifs de tests unitaires (ex. : JUnit). Concernant les Mocks, il est de plus en plus rare que ceux-ci soient créés à la main étant donné l'utilisation extensive prônée par les agilistes. Les équipes emploieront donc des cadres applicatifs de Mocks virtuels (ex. : JMock [jmo] ou Mockito [moc]) permettant de générer dynamiquement ces classes (les objects factices).

Nous pensons donc qu'il est important de proposer une solution se rapprochant le plus possible des solutions déjà existantes, employées et connues en OO. Nous croyons que cela permettra une meilleure adoption, diminuera le temps d'apprentissage et facilitera l'intégration. Et ce faisant, augmentera la productivité des équipes ainsi que la stabilité [fSI01] du logiciel testé.

Nous croyons également qu'il est important que le cadre applicatif s'intègre facilement au processus de test, de développement et de déploiement afin de minimiser les irritants, les duplications et les surprises à tous les niveaux du processus. Étant donné l'omniprésence de l'OO et le nombre potentiellement élevé d'objets, les équipes préféreront possiblement ne pas utiliser l'AO plutôt que de devoir remplacer ou modifier substantiellement le système de tests et de déploiement déjà en place dans leurs projets.

Finalement, nous pensons que, considérant les contraintes de temps et de budget dans les entreprises, le fait d'utiliser une solution clés en main, fonctionnelle, facile d'utilisation et s'approchant de ce qui est déjà utilisé, facilitera le travail des équipes devant convaincre leurs patrons et gestionnaires de projet d'adopter l'AO ; ceci en raison de la réduction du temps d'apprentissage et de l'effort moindre requis pour intégrer la technologie dans le processus et avec les outils existants. Ainsi, le ROI [2] lié à l'utilisation de l'AO risque d'être meilleur du point de vue de l'entreprise.

En définitive, il faut également considérer qu'une équipe agile risque, idéalement, d'adopter une technologie uniquement s'il est possible d'en automatiser les tests. De plus, plusieurs équipes agiles font des réusinages continuels, utilisent souvent des intégrateurs en continu et effectuent des déploiements fréquents. Nous pensons donc que ces équipes bénéficieront, encore plus, d'avoir une technologie de tests s'intégrant facilement aux technologies existantes d'automatisation et permettant de réaliser des tests unitaires d'aspects répondant aux règles de l'art.

1. Rappelons que nous étudions les projets s'inscrivant dans le cadre d'un processus Agile.
2. Retour sur investissement

4.1.3 Objectifs

L'objectif général est de répondre au besoin relevé dans l'étude (voir chapitre 3) concernant la facilité de réalisation des tests d'aspects. Par conséquent, notre but est d'augmenter la testabilité des logiciels industriels utilisant de l'AO et ce faisant, leur qualité (stabilité, modifiabilité, etc.) [fSI01].

Solution intuitive et naturelle pour un développeur OO.

Pour ce faire, nous proposons de développer un cadre applicatif (framework) permettant de tester unitairement les aspects en isolation via la technique des Mocks d'objets déjà largement utilisée et qui a fait ses preuves en OO. Nous tenterons, le plus possible, de l'intégrer aux outils existants et déjà utilisés dans le monde Java industriel. De plus, nous accorderons une importance particulière à l'utilisabilité [fSI01] afin d'en rendre l'emploi le plus naturel possible pour des développeurs familiers avec les tests d'objets.

Nous concevrons ce cadre applicatif de manière à ce qu'il soit intuitif pour des personnes familières avec les concepts d'AO et AspectJ en reprenant l'ontologie et la philosophie associées. Cela tout en nous arrimant le plus possible avec les techniques de tests OO existantes et leur ontologie.

Solution opérationnelle et facilement intégrable.

Nous visons à développer une application fonctionnelle de qualité production et utilisable telle quelle. De plus, afin de faciliter son intégration aux outils existants, le cadre applicatif devrait être une solution 100% pur Java compatible avec une version de Java ≥ 5. Ceci implique :

- d'offrir toutes les fonctionnalités via un API Java ;
- de ne pas recourir à des scripts, commandes ou outils externes ;
- d'être indépendant de l'IDE (ex. : Eclipse) ;
- de ne pas utiliser d'autres langages ou extensions à Java (donc pas d'AspectJ) ;
- de fonctionner sur toutes les JVMs compatibles J2SE 5.0 (JSR-176[jsr04]).

De plus, la solution doit pouvoir s'intégrer facilement à des cadres applicatifs de Mocks existants et avec JUnit [jun] version 4.

Pour ce qui est d'AspectJ, nous désirons offrir une solution fonctionnant avec la version officielle actuelle. Cela implique de n'effectuer aucune modification au code d'AspectJ et de l'utiliser tel quel.

Solution extensible.

Le but est d'offrir un cadre applicatif extensible permettant facilement l'ajout et le support de nouvelles fonctionnalités tout en respectant le principe de simplicité demandé dans les projets agiles. Le but n'est pas d'implanter tous les cas possibles et de couvrir toutes les possibilités offertes par AspectJ mais plutôt d'implanter les cas les plus courants tout en permettant une évolution aisée par la communauté.

Convivialité (utilisabilité) [3]

Facilité d'utilisation. Étant donné qu'il s'agit d'un cadre applicatif de tests unitaires, les utilisateurs visés sont les développeurs. Par conséquent, l'interface de la solution doit s'y adapter afin de répondre aux besoins d'une catégorie d'utilisateurs experts.

Premièrement, l'interface en sera une de programmation (API) et non pas graphique. Aucune action manuelle ne doit être requise et il doit être possible d'utiliser cette interface à partir de tests JUnit version 4 [jun]. Elle doit respecter les conventions de Java de nommage et de design pour les APIs.

Finalement, pour faciliter l'utilisation dans un projet Java standard, le cadre applicatif doit être fourni sous la forme d'une archive Java standardisée (JAR). Étant donné que le cadre utilisera plusieurs dépendances qui risquent d'être employées dans d'autres cadres utilisés conjointement (comme JUnit ou le cadre pour les Mocks), il sera nécessaire d'offrir une certaine flexibilité au développeur pour réutiliser ou non la dépendance conjointe. Nous devrons donc [4] :

- fournir un JAR contenant uniquement notre cadre ;
- fournir un autre JAR contenant notre cadre ainsi que toutes les dépendances ;
- permettre la résolution automatique des dépendances via Maven[Fou].

Facilité de compréhension. Une grande complexité est associée à la génération automatique de classes, au tissage dynamique et à la modification à la volée de code binaire (Bytecode Java). Étant donné que nous combinons plusieurs technologies n'étant pas conçues initialement pour fonctionner de pair, nous aurons à combiner plusieurs modèles et vocabulaires. Par exemple, le même type d'erreur dans AspectJ et dans CGlib ne porteront pas le même nom car ils n'opèrent pas au même niveau.

Par conséquent, nous devrons accorder un soin particulier à cacher cette complexité et cette diversité à l'utilisateur. Nous devrons lui donner des rétroactions cohérentes et harmoniser le vocabulaire.

Dans le même ordre d'idées, les raisons pour lesquelles un test échoue devront être claires et devront être présentées dans le vocabulaire du test et non pas celui de la technique utilisée pour

3. Les caractéristiques sont tirées de la norme internationale ISO-9126 [fSI01]
4. Il s'agit là d'une pratique répandue et presque standard dans la communauté Java.

procéder au test. De plus, les raisons pour lesquelles un test sur un aspect échoue peuvent parfois être conceptuellement complexes en raison de l'injection, des points de coupures et des interactions et effets de l'aspect. Il sera donc important que la raison de l'échec soit présentée de la manière la plus évidente que possible en offrant un texte contextualisé, aisément compréhensible et offrant des détails.

Finalement, l'interface doit faire en sorte de faciliter la lecture du test même pour un développeur n'ayant jamais utilisé notre cadre applicatif. En effet la lisibilité d'un test permet à ce dernier d'agir en tant que documentation fonctionnelle, caractéristique recherchée dans les projets Agiles.

Facilité d'apprentissage. L'interface devra être facile à apprendre pour un développeur déjà expérimenté et habilité à réaliser des tests d'objets à l'aide de Mocks et de JUnit.

Attractivité. L'interface proposée doit être simple et naturelle afin d'inciter les développeurs à écrire des tests le plus proprement possible pour les aspects. Cela pourrait également permettre une meilleure ou encore une adoption plus large de l'AO au sein de projets.

4.1.4 Démarche

Pour y arriver, nous procéderons comme suit :

1. Développer un prototype fonctionnel afin :
   - d'éliminer les risques ;
     - difficultés à tisser des classes générées dynamiquement ce qui pourrait demander d'importantes modifications à AspectJ [prototype A] ;
     - difficultés à isoler les aspects et à les sélectionner en raison de limitations dans AspectJ [prototype A] ;
     - difficultés à simuler à l'exécution un contexte qui n'existe pas réellement (un package, un nom de classe ou méthode, etc.) [prototypes B et C] ;
     - incapacité à offrir une interface simple à l'utilisateur et compatible avec les outils de l'IDE (réusinage, auto-complétion, etc.) en raison des limites de la réflexivité et de la nature statique du langage Java ;
     - incompatibilité entre AspectJ et les cadres applicatifs de Mocks [prototype A] ;
     - interactions non désirées entre le compilateur et notre cadre applicatif [prototype A] ;
     - limitation de la JVM ;
     - limitations reliées à la génération de classes dynamiques à l'exécution.
   - d'évaluer la faisabilité ;

- de tester la compatibilité avec Mockito [moc] et JUnit 4 ;
- d'adapter le tisseur (AspectJ Weaver) conçu pour le LTW [5] afin de pouvoir démarrer dynamiquement dans un test et avec un contexte restreint ;
- de supporter les classes dynamiques et « proxies » [6] puisqu'il sera nécessaire d'utiliser ces technologies pour répondre à notre besoin ainsi qu'en raison de l'utilisation de cette technique par la majorité des cadres applicatifs de Mocks virtuels modernes ;
- de tenter de prouver qu'il est possible de réaliser une solution 100% pure Java sans utiliser de Java Agents (voir section 5.1.5).

2. Développer le cadre applicatif :
   - définir la spécification ;
   - concevoir une architecture flexible, extensible et simple ;
   - implanter les fonctionnalités requises.

Pour ce faire, nous adoptons un processus Agile inspiré de Scrum avec des sprints de 2 semaines. Chaque fonctionnalité est découpée en scénarios comme le montre la section A. Bien entendu, les fonctionnalités seront testées et documentées.

Finalement, il est possible d'évaluer la solution proposée à partir des tests d'acceptation utilisateur élaborés. Cela permet de s'assurer que la solution répond aux attentes établies en fonction des contraintes et objectifs (voir section 4.1.3) ainsi qu'au besoin initial (section 4.1.1 et chapitre 3).

4.2 Démarche et approche menant à la solution

La solution proposée repose sur l'adaptation des techniques déjà utilisées, répandues et adoptées pour les tests unitaires d'objets. Nous proposons une solution simple pour l'utilisateur lui permettant de réutiliser les technologies déjà présentes dans un projet Java objet.

Puisque nous nous intéressons au projet se déroulant dans un contexte Agile, nous devons d'abord extraire les bonnes pratiques recommandées dans ce contexte pour un projet OO. Tel que présenté à la section 2.5.2, la littérature industrielle Agile recommande l'utilisation de doublures permettant d'isoler l'objet testé. La section 2.5.3 a montré le fonctionnement d'une catégorie de doublures que l'on nomme « Mock » et pour lesquelles une panoplie de cadres applicatifs existent pour le monde objet.

Étant donné ce contexte, il semble intéressant de se demander si une telle technique est applicable et pourrait permettre de tester de manière similaire des aspects. Pour ce faire, analysons

5. Load-Time Weaving (tissage dynamique à l'exécution).
6. Ces classes n'ont pas de code source et sont générées (le « ByteCode ») dynamiquement à l'exécution et chargées par le chargeur de classes (classloader). Ces classes sont généralement générées avec des technologies comme ASM, CGLib, etc.

la logique et les motivations de cette technique en OO et tentons d'établir un parallèle avec l'AO.

4.2.1 Parallèle entre les tests unitaires en OOP et AOP

Tel que décrit à la section 2.5.3, le but d'un test unitaire est de tester la classe sous-test en l'isolant du reste du système.

Tests unitaires d'objets

Pour ce faire, le test pilotera la classe testée (typiquement en y appelant des méthodes) et vérifiera les *effets* produits (les états et les interactions).

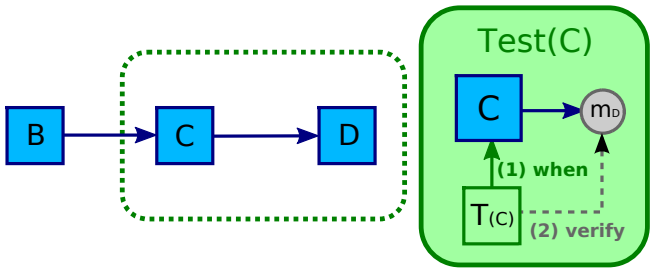


FIGURE 4.1: Test unitaire OO : isolation d'une classe.

Remarquons qu'un test unitaire ne se préoccupe *pas* des classes qui *ont une dépendance* vers la classe testée *(afférent)*. Uniquement les *dépendances de celle-ci (efférent)* sont considérées afin de vérifier si la classe testée a eu les interactions (comportement) espérées. D'ailleurs ce sont ces dépendances efférentes qui seront remplacées par des Mocks alors que les dépendances afférentes sont remplacées, en quelques sorte, par le test.

En bref, dans un test unitaire, seules les dépendances efférentes (à partir de la classe testée) sont considérées comme le montre les figures 4.1 et 4.2. Sur ces schémas, « C » est la classe testée, « D » une dépendance efférente de « C » et « B » est une classe utilisant « C » (dépendance afférente).

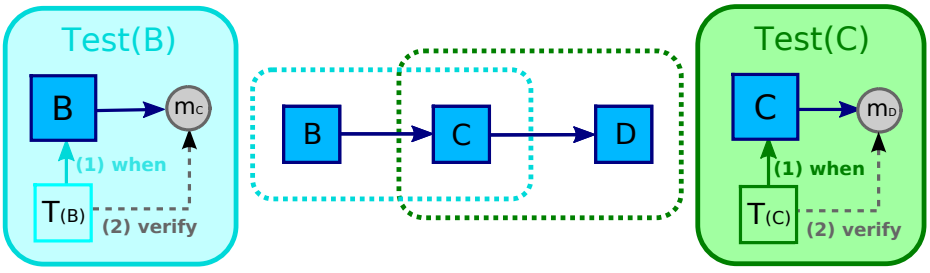


FIGURE 4.2: Test unitaire OO : décomposition des tests.

Tests unitaires d'aspects

Afin de tester unitairement un aspect, il faut tester deux choses [Les05] :

- les comportements produits (fonctionnalité des greffons et méthodes) ;
- où s'appliquent les comportements (spécification des points de coupure).

Le premier cas est très similaire aux tests unitaires d'objets à l'exception qu'il n'est pas possible d'invoquer directement un greffon. Mais d'un point de vue logique, il s'agit fondamentalement de la même chose qu'une méthode.

Avec un aspect, la responsabilité de décider où et *quand un greffon doit s'exécuter incombe à l'aspect lui-même*. Par conséquent, il faudra tester, dans le test de cet aspect, que ses greffons s'exécutent au bon moment.

Dans un test unitaire objet, il n'est pas nécessaire de vérifier quand les méthodes de la classe testée sont appelées puisqu'il s'agit de dépendances afférentes [7]. Par exemple, à la figure 4.2, la responsabilité d'appeler « C » incombe à la classe « B ». Par conséquent, l'appel de « C » sera plutôt testé dans le test « T(B) ».

En comparaison, la figure 4.3 montre clairement que « l'appel » d'un greffon dans « A » devra être testé dans « T(A) ». Contrairement à une classe objet, ça ne peut être fait dans le test de « E » puisque « E » n'est pas conscient de l'existence de « A » et qu'il est de la responsabilité de « A » de décider quand ses greffons doivent être exécutés.

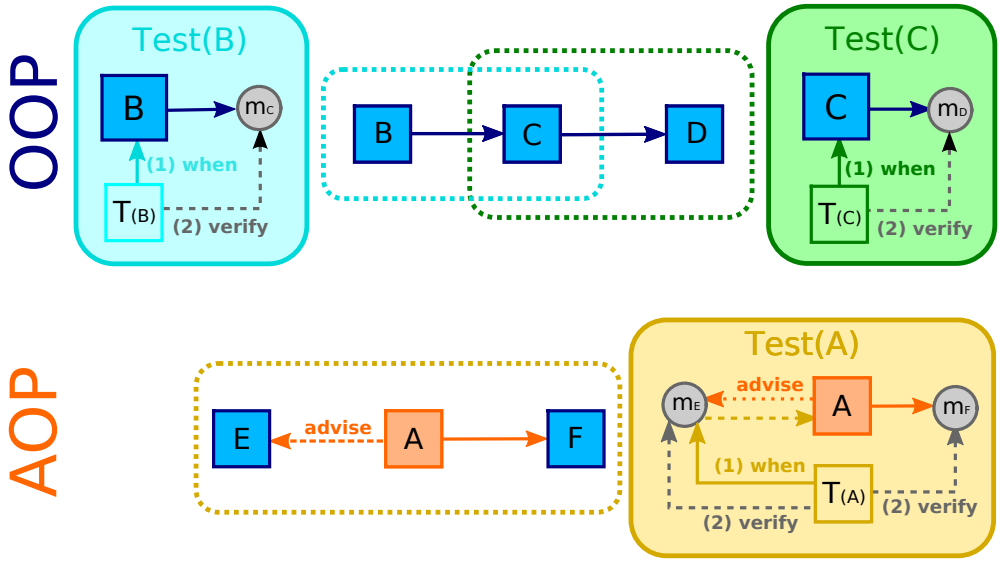


FIGURE 4.3: Tests unitaires d'objets ou d'aspects.

Concernant la technique à employer pour réaliser ces tests, nous verrons à la section 4.2.3 comment les Mocks peuvent offrir une solution intéressante.

7. En OO, un objet ne sait pas quand l'une de ses méthodes sera appelée. Bref, une classe n'est pas consciente des autres classes qui dépendent d'elle (dépendances afférentes).

4.2.2 Caractéristiques des tests unitaires d'aspects

Isolation. Tout comme les classes, un aspect regroupe une préoccupation qui doit être testée. Puisque l'objectif d'un test unitaire est de tester le comportement d'une unité en isolation et qu'une unité devrait, théoriquement, n'avoir qu'un seul but (SRP [Mar02c]), on peut en conclure qu'*un test unitaire teste une préoccupation en isolation.*

Considérant le fait que l'objectif même de l'*AO est de séparer les préoccupations transverses*, il paraît alors évident que le principe d'isolation des tests unitaires reste pertinent avec des aspects et, peut-être même, à plus forte raison qu'avec des objets.

Dépendances. La principale différence se situe plutôt dans la manière dont les comportements sont déclenchés (le « when » dans les tests unitaires) :

- par des appels aux *méthodes* (y compris les injections – ITD) s'il y en a ;
- par l'appariement (matching) d'un point de jonction qui déclenche l'exécution d'un *greffon.*

Le premier cas est très similaire aux classes à l'exception de l'instanciation qui peut souvent s'assimiler à un objet singleton dans le monde OO. La technique de test sera donc très proche de celle employée avec des méthodes dans une classe.

Le cas de l'invocation des greffons constitue donc la principale différence avec les tests d'objets puisque c'est l'aspect qui décide, lui-même, quand et où les greffons s'exécuteront via les points de coupure (pointcuts). En effet, c'est lors de la rencontre d'un point de jonction apparié, telle l'exécution d'une méthode, par exemple, que se déclenchera le comportement espéré sur l'aspect.

De par la nature des préoccupations transverses, la responsabilité de décider à quel moment doit s'exécuter un greffon incombe à l'aspect lui-même. Puisqu'il s'agit d'une *responsabilité de l'aspect, c'est au test unitaire de cet aspect qu'incombe la tâche de tester ce comportement.*

Puisque les classes visées par des points de coupures d'un aspect ne sont pas conscientes de ce dernier, nous pouvons considérer que le point de coupure est une dépendance efférente de l'aspect et non l'inverse. Du point de vue des tests, cette vision se tient puisqu'ils seront testés dans le test de l'aspect comme le montre la figure 4.3.

Par conséquent on peut en déduire :

- que les *classes tissées sont une forme de dépendance* efférente de l'aspect et non l'inverse ;
- *que le test devra provoquer le comportement à tester sur l'aspect en « activant » un point de jonction.*

4.2.3 Utilisation de Mocks en AOP

Dans le cas de tests d'aspects, nous pourrons utiliser les Mocks dans deux cas :

- afin de tester le comportement (la logique) de l'aspect (section 4.2.3) ;
- afin de tester où (point du coupure) ce comportement doit s'exécuter (section 4.2.3).

Tester les comportements espérés et la logique

Comme les classes, les aspects ont des comportements qui doivent être testés. Ces comportements peuvent prendre différentes formes (méthodes, greffons, etc.). Cependant, peu importe leur forme, l'objectif du test est de vérifier que le comportement résultant est bien celui espéré.

Tout comme pour les tests d'objets, nous voudrons tester les effets produits (interactions) par l'aspect sur d'autres classes. En d'autres mots, nous désirons tester les messages (méthodes) envoyés vers des modules externes. Tel qu'expliqué à la section 2.5.3, nous utilisons généralement à cette fin des Mocks qui permettent à la fois d'enregistrer les appels (vérifications) mais également de fixer le comportement des modules externes afin de placer l'unité testée dans un contexte spécifique au test courant.

En ce sens, un aspect fonctionne exactement comme une classe car la logique contenue dans ses greffons et méthodes est de la programmation OO standard.

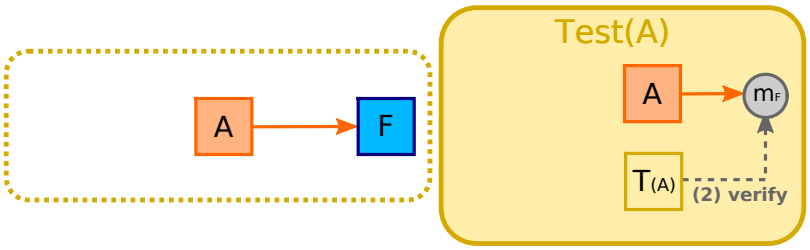


FIGURE 4.4: Tests unitaires d'aspects : comportements espérés

Par conséquent, il sera possible de tester le flot d'appels émanant de la logique de l'aspect de la même manière qu'avec des classes en employant des Mocks sur lesquels nous ferons des vérifications comme le montre la figure 4.4. Il faut cependant considérer le fait qu'un aspect n'a pas le même cycle de vie (instanciation) et que par conséquent, il faudra :

- adapter la méthode d'injection des Mocks ;
- prévoir de réinitialiser l'aspect entre les tests.

En fait, dans la majorité des cas, l'aspect se comportera comme un singleton et les techniques associées s'appliqueront généralement. Diverses techniques existent, dont l'emploi d'aspects pour injecter les Mocks dans d'autres aspects, mais celles-ci dépassent le cadre de ce chapitre, notre objectif de recherche ne se situant pas à ce niveau.

En résumé, l'utilisation de Mocks afin de vérifier les comportements de l'aspect se fera de manière très similaire aux tests de classes en employant des techniques déjà connues.

Activation des comportement (exécuter un greffon)

La grande différence avec les aspects, du point de vue des tests unitaires, réside dans leur capacité à spécifier quand et où les comportements (greffons) seront exécutés. En fait, les greffons ne sont *jamais exécutés directement* dans des circonstances normales.

Afin de déclencher l'exécution d'un greffon dans un test, l'idéal est d'utiliser le même mécanisme que le code réel de production utilisera [8]. Pour ce faire, nous pouvons :

1. créer un cas (un Mock) simulant un contexte où le point de coupure devrait apparier (« given ») ;
2. tisser l'aspect à tester dans le Mock ;
3. exécuter ce cas (« when ») sur le Mock (le déclencheur) ;
4. finalement, nous pourrons vérifier si le résultat attendu est obtenu (« then ») suite à l'exécution du ou des greffons supposés s'exécuter suite à ce stimulus.

En effet, si *le comportement est obtenu suite à l'exécution (rencontre) d'un point de jonction*, cela signifie que *le point de coupure a apparié* et que *le bon greffon s'est exécuté*.

Dans un aspect, un greffon s'exécute quand un point de jonction est apparié par un point de coupure. En d'autres termes, l'aspect définit un point de coupure qui spécifie où un ou des greffons s'exécuteront.

Ainsi, pour déclencher l'exécution d'un greffon, il faut simplement exécuter un code contenant un point de jonction apparié par un point de coupure associé à ce greffon.

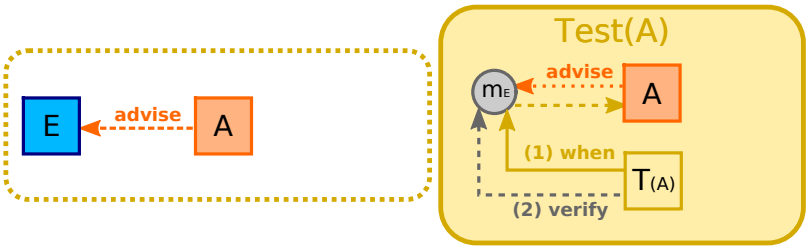


FIGURE 4.5: Tests unitaires d'aspects : activation (cible des points de coupures).

La figure 4.5 montre comment un Mock peut être utilisé afin de déclencher le comportement désiré sur l'aspect. Des vérifications seront ensuite possibles sur ce même Mocks comme nous le verrons dans les prochaines sections.

Vérifier la portée (cibles) des points de coupure.

Précédemment, nous avons expliqué comment il est possible de déclencher l'exécution d'un greffon à l'aide d'un Mock. Par ricochet, nous avons été en mesure de vérifier que le point de

8. Nous traitons de test unitaires et non pas de tests fonctionnels ou d'intégration. Par conséquent, nous voulons reproduire l'*utilisation réelle de l'unité* testée en isolation mais non pas de l'ensemble des unités voisines.

coupure pouvait apparier au moins dans un contexte donné et ainsi provoquer l'exécution du greffon.

Par contre, ce résultat n'est pas suffisant puisque nous ne savons pas si la portée du point de coupure est adéquat. En d'autres mots, nous devons aussi vérifier si le point de coupure fonctionnera ou non avec différents points de jonctions (différents contextes). Nous voulons donc tester que le point du coupure :

- apparie suffisamment de points de jonction ;
- n'apparie pas plus de points de jonction qu'il ne le devrait.

Exemple 6

Si nous avons le point de coupure suivant : « `execution( void doX*() )` », nous pourrions, par exemple, vouloir vérifier les cas suivants :

Méthode	Résultat	Méthode	Résultat
`void doX()`	OUI	`int doX()`	NON
`void doXY()`	OUI	`void doX(int)`	NON
`void doY()`	NON		
`void do()`	NON		

Remarquons que les cas de test de l'exemple 6 sont montés de la même manière que pour les tests de méthodes classiques. On peut y employer des techniques de construction connues comme les valeurs limites, classes d'équivalence, etc.

Technique des pots de miel (Mock Targets [Les05]). Afin de tester la portée des points de coupure la communauté utilise parfois une technique appelée pots de miel (honeypots) ou Mocks Targets. Cette technique a, notamment, bien été décrite par Nicholasdans Lesiecki dans [Les05].

Cette technique consiste à créer une fausse classe, d'y tisser l'aspect testé et, finalement, de vérifier si les points de coupures ont apparié uniquement là où ils le devaient. Il s'agit, en fait, de créer une classe avec des appâts permettant de vérifier les cas de tests.

Bref, cette *fausse classe n'est rien d'autre qu'un Mock* à la différence près qu'un Mock Object standard reçoit les appels de la classe testée alors qu'ici le *Mocks Target* vise plutôt à *stimuler l'aspect testé* en servant de *« cible » pour les points de coupures.*

Nicholasdans Lesiecki résume cette idée dans ces termes :

« I introduce a term I invented to describe a type of test helper that is useful in writing aspect tests: mock targets. In the pre-aspect world, a mock object denoted a class [...] that imitated a collaborator for some class you were attempting to test. Similarly, a mock target is a class that imitates a legitimate advice target for some aspect you are attempting to test. [Les05] »

Exemple 7

Reprenons l'exemple 6 avec le point de coupure suivant : « `execution( void doX*() )` ».

Dans ce cas, nous pourrions créer un Mock Target possédant les méthodes suivantes :

```
interface MockForExecDoX {
    void doX();
    void doXY();
    void doY();
    void do();
    int doX();
    void doX(int);
}
```

Ces méthodes s'avèrent correspondre directement aux cas de test de l'exemple 6.

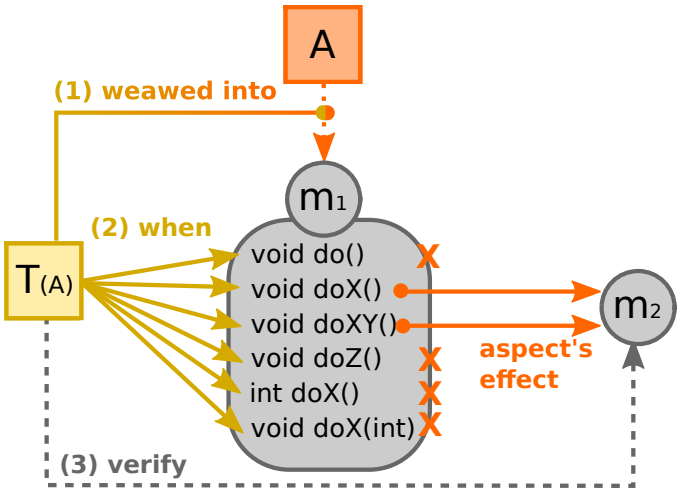


FIGURE 4.6: Tests unitaires d'aspects : utilisation d'un Mock pour vérifier les points de coupure.

Par la suite, le test pourra appeler les méthodes sur le Mock (« when ») et ainsi vérifier (« then ») lesquelles déclenchent l'exécution des greffons comme le montre la figure 4.6. Cette vérification est très similaire à la vérification des méthodes appelées par la classe testée sur un Mock Object classique.

Notons qu'utiliser un Mock, rend possible l'utilisation de ces derniers à la fois comme cible et comme récepteur (Mock Target + Mock Object) puisqu'il n'est pas rare qu'un greffon produise un effet sur le même objet ciblé par le point de coupure. Le Mock nous permettra donc de déclencher l'événement puis de vérifier les appels résultants comme le montre l'exemple 8.

Exemple 8

Aspect testé :

```
public aspect AspectUnderTest {
    pointcut pc(Target t) :
        execution( void Target+.doX*() )
        && this(t);

    after(Target t) : pc(t) {
        t.callByAspect();
    }
}
```

Interface existante :

```
interface Target {
    void doX();
    void callByAspect();
}
```

Test (partiel) :

Note : Ce test est partiel et ne répond pas aux règles de l'art en matière de rédaction de tests.

```
interface MockTarget extends Target {
    void doXY();
    void doZ();
} // Test partiel...

@Test public void whenPCMatch_thenCallByAspectIsCalled {
    // given
    MockTarget mock = mockWeaved(MockTarget.class,
                                 AspectUnderTest.class);
    // when
    mock.doX(); // OUI
    mock.doXY(); // OUI
    mock.doZ(); // NON
    // then
    verifyCalled(mock.callByAspect(), times(2));
}
```

Dans ce cas, l'objet « `mock` » sert, à la fois, afin de stimuler l'aspect et pour vérifier le comportement attendu de ce dernier.

4.2.4 Résumé

En résumé, la stratégie de test que nous employons afin d'élaborer un test unitaire d'aspect en isolation se résume comme ceci :

1. Créer et injecter des Mocks pour toutes les dépendances de l'aspect comme nous le ferions afin de tester une classe objet.
2. Créer un ou des Mocks agissant comme cibles pour les points de coupures de l'aspect. Ces Mocks contiennent des points de jonctions devant être appariés ainsi que d'autres similaires mais ne devant pas l'être.
3. Tisser l'aspect dans les Mocks servant de cibles puis exécuter ces cibles.
4. Vérifier le comportement de l'aspect pour les différents cas à tester.

Notre solution (présentée à partir de la section 4.2 et au chapite 5) se basera sur cette technique de test pour laquelle il n'existe pas de cadre applicatif permettant l'automatisation de ces tests. De plus, comme nous le verrons aux section subséquentes, plusieurs cas sont pratiquement impossibles à tester sans une solution informatisée permettant la création dynamique de classes en mémoire et le plein contrôle afin de sélectionner les aspects à tisser pour chacun des tests.

Notre solution permettra au développeur qui maîtrise déjà la technique des Mocks en OO, de tisser ces Mocks et d'en générer d'autres pour des contextes spécifiques aux tests d'aspects.

4.2.5 Discussion

Séparation des points de coupure et des greffons

Certains préfèrent tester indépendamment l'appariement des points du coupure et les greffons. Nous avons déjà établi (voir section 4.2.1) le fait que la spécification des points de coupure fait partie intégrante des responsabilités de l'aspect. Par conséquent, puisqu'un test unitaire vise à tester de l'extérieur l'unité sous-test, il nous paraît logique de tester, au moins dans un test, le comportement du greffon via un point de jonction apparié.

Avec l'approche que nous préconisons, l'aspect est utilisé tel qu'il le sera par le code de production ce qui rend le test plus réaliste. Séparer les deux composantes de l'aspect fait en sorte que l'on teste, en quelque sorte, une version partielle de l'aspect qui est pourtant l'unité à tester.

De plus, dans certains cas, tester indépendamment les points de coupure et les greffons est plus simple car il ne requiert pas forcément de tisser l'aspect. C'est précisément parce que ce contrôle du tissage est ardu que nous proposons notre solution qui permet précisément d'isoler l'aspect et de contrôler son tissage dans les Mocks.

Finalement, il est parfois recommandé [Les05] d'extraire la logique des aspects afin de respecter une meilleure séparation des préoccupations. Ceci permet, notamment, de respecter le SRP [Mar02c] appliqué aux aspects [Wam07]. Une fois la logique extraite dans une classe

utilisée par les greffons, il subsiste encore moins d'avantages à tester les points de coupure et les greffons séparément puisque ces derniers ne possèdent désormais que peu de logique. Dans ce contexte, la ou les classes possédant la logique seront des dépendances de l'aspect et, par conséquent, représentées par des Mocks dans le test.

En résumé nous préconisons une approche où l'on vérifie que l'aspect sous-test :

- s'exécutera au bon moment ;
- et qu'uniquement le ou les bons greffons seront exécutés à ce moment.

Il peut cependant exister des cas et de bonnes raisons pour vouloir, parfois, séparer ces cas dans le test. Ceci n'est pas en contradiction avec la technique que nous proposons et peut être aisément ajouté comme *technique complémentaire*.

Couverture du test

La technique proposée permet d'obtenir, en théorie, des métriques de couverture très élevées, voire complètes des greffons.

Par contre, elle ne garantit pas que toutes les cibles possibles pour les points de coupure ont été testées. En fait, dans un langage comme AspectJ permettant des patrons (wildcards), cela est pratiquement impossible en raison du nombre potentiel de tests à réaliser par un développeur humain.

Le but de nos travaux est de trouver un moyen de porter les techniques de tests unitaires OO vers l'AO. Le but de nos recherches n'est pas de développer un outil de vérification formel permettant de prouver les spécifications d'un aspect. Nous souhaitons plutôt, à ce stade, fournir à l'industrie un moyen de tester, de manière toute aussi efficace leurs systèmes Java (OOP) et AspectJ (AOP) dans un contexte industriel complexe impliquant de nombreuses technologies et méthodologies.

Chapitre 5

Mock4Aj : Solution proposée

Mock4Aj est un cadre applicatif qui permet d'écrire des tests unitaires d'aspects en isolation. Le code du projet est ouvert sous licence LGPL et disponible en téléchargement : `mock4aj.googlecode.com`

5.1 Défis et innovation

L'objectif principal du cadre applicatif que nous développons est d'automatiser et d'offrir un outil convivial permettant aux développeurs de tester un aspect en isolation à l'aide de Mocks comme nous l'avons déjà établi au chapitre 4.

De manière générale, les principaux défis et innovations sont :

- isoler l'aspect ;
- tisser des classes dynamiquement générées (les Mocks virtuels) ;
- simuler des contextes qui n'existent pas réellement (ex. : un nom de méthode qui n'existe pas) par la création de Mocks sur mesure ;
- d'offrir une interface programmable (API) rendant possible l'utilisation des outils de réusinage (refactoring) ainsi que la complétion automatique offerte par les IDEs modernes ;
- de fonctionner sur une JVM standard, sans utiliser de Java Agent, en s'intégrant facilement avec d'autres outils de Mocks et sans entrer en conflit avec d'autres technologies courantes.

5.1.1 Isolation de l'aspect

Afin d'effectuer un test réellement unitaire, il faut d'abord être en mesure d'isoler l'aspect :

- des classes dépendantes ;
- des classes dont il dépend ;
- *des autres aspects.*

Les deux premiers cas sont triviaux en raison de l'utilisation de Mocks et de la nature du test comme nous l'avons déjà vu à la section 4.2.3. De plus, ils sont quasi identiques aux tests unitaires objet.

Par conséquent, le cadre applicatif n'offrira aucune fonctionnalité pour ces deux cas. Cependant, nous avons veillé à ce qu'il s'utilise conjointement et n'entre pas en conflit avec les cadres applicatifs de Mocks tiers déjà employés dans l'industrie (ex. : Mockito).

L'isolation d'un aspect des autres aspects présents dans le système est beaucoup plus complexe [1]. Ceci est intrinsèquement causé par le fonctionnement même d'AspectJ. Il y a deux cas de figure à considérer en fonction du mode de tissage :

- tissage à la compilation (CTW : Compile Time Weaving) ;
- ou tissage à l'exécution (LTW : Load Time Weaving).

Isolation en CTW. En CTW, les aspects sont *tissés lors de la compilation du code source.* Évidemment, dans ce contexte, il est strictement *impossible de contrôler quels aspects seront tissés à partir d'un test (à l'exécution)* puisque la classe a déjà été compilée, et donc tissée.

De plus, AspectJ n'offre aucun mécanisme à ce jour pour restreindre le tissage des aspects (sans modifier l'aspect lui-même) par annotation statique ou autre pouvant s'appliquer dans notre cas.

Étant donné le fonctionnement d'AspectJ en CTW, le code (le Bytecode) de la classe tissée est altéré afin d'y injecter l'appel vers l'aspect. Il n'est donc pas aisément possible de revenir en arrière une fois la classe compilée et l'aspect tissé.

Dans notre cas, nous désirons tisser des Mocks virtuels qui sont, en fait, dynamiquement générés pendant l'exécution. Ces classes n'existent pas réellement et ne sont jamais compilées à la phase de compilation puisque le Bytecode est directement chargé en mémoire.

Cette particularité s'avère très utile car elles passeront sous le radar du tisseur qui ne les tissera pas. Elles se trouvent donc naturellement isolées. Il nous sera donc possible de contrôler quel aspect nous désirons tisser dans chacun des Mocks lors de l'exécution comme nous le verrons à la section 5.2.2.

1. Cette complexité peut varier en fonction du tisseur et du langage utilisé. Dans notre cas, nous nous concentrerons sur l'utilisation d'AspectJ.

AspectJ en CTW n'est donc pas un obstacle dans notre cas à condition d'utiliser notre cadre applicatif exclusivement sur des classes dynamiques (comme les Mocks virtuels). Mock4Aj accepte de travailler sur des objets qui ne sont pas des classes dynamiques pour des cas d'utilisation avancés. Cependant, l'utilisateur est averti qu'il s'agit d'une utilisation risquée.

Isolation en LTW. AspectJ offre également un deuxième mode de tissage permettant des *tisser les aspects lors de l'exécution.* Pour être plus précis, le tissage est alors effectué lors du chargement des classes (Class Loading) par la JVM. AspectJ intercepte le Bytecode original, le transforme et c'est ce Bytecode tissé qui est alors chargé par le chargeur. Ce mécanisme de transformation est rendu possible par la JVM via un mécanisme nommé « *Java Agent* » [2].

Une fois ce Java Agent en place, il n'existe pas de moyen standard [3] d'obtenir une référence sur ce dernier afin d'altérer sa configuration dynamiquement à partir du code du test. Il serait possible d'utiliser certaines voies de contournement (ex. : modifier les fichiers aop.xml d'AspectJ à la volée) mais ces dernières relèveraient de la rustine (patch), ne seraient pas solides, n'auraient aucune garantie de fonctionner dans l'avenir et seraient difficilement utilisables par l'utilisateur puisque AspectJ en LTW n'a pas été conçu pour cela.

Puisque le tissage en LTW survient lors du chargement des classes à l'exécution, les classes générées comme les Mocks virtuels, seraient automatiquement tissées lors de leur chargement. C'est clairement un comportement souhaitable dans notre cas si ce n'était que *nous ne sommes pas en mesure de contrôler quels aspects seront tissés.* En fait, tous les aspects seront tissés s'ils ont un point de jonction apparié.

Par conséquent, *pour que Mock4Aj fonctionne correctement, il est impératif que le LTW soit désactivé lors de l'exécution des tests.* Sans quoi, l'aspect testé ne sera plus isolé et pourrait même être tissé plusieurs fois.

Notons toutefois que notre cadre applicatif utilisera une forme modifiée de LTW pour fonctionner comme nous l'expliquerons à la section 5.2.2. De plus, notre solution ne requiert pas l'utilisation d'un Java Agent ce qui présente des bénéfices tels qu'expliqués à la section 5.1.5.

2. Notons qu'il existe également un autre mécanisme qui consiste à changer le chargeur de classe (Class Loader) mais cette méthode est de moins en moins répandue depuis l'avènement de Java 5.

3. Cette fonctionnalité dépend de chaque implémentation de JVM. Cette technique ne peut donc pas fonctionner dans tous les cas et n'est pas portable.

Résumé. On peut résumer le problème d'isolation comme ceci :

- Le tissage à la compilation (CTW) :
  - *ne permet pas de tisser un Mock ;*
  - le cadre applicatif *peut donc sélectionner quel aspect tisser (isolation) sans avoir de conflit* mais il faut trouver une autre solution pour faire le tissage.
- Le tissage à l'exécution (LTW) :
  - *peut tisser dans les Mock* virtuels ;
  - *tisse toujours tous les aspects* (pas d'isolation) configurés ;
  - *tisse dans tous les Mocks* (impossible d'écrire un cas de test où l'aspect n'est pas appliqué) [4].

En conclusion il faut :

- désactiver le LTW par défaut de AspectJ pour obtenir une isolation des aspects ;
- concevoir un mécanisme similaire au LTW mais permettant la sélection des aspects à tisser ainsi que des Mocks dans lesquels les tisser (voir 5.1.2).

Références techniques:

- *Scénarios utilisateurs liés :* M4AJ-01a, M4AJ-01b, M4AJ01c, (voir annexe A).
- *Test d'acceptations associés :*
  - TestAnAspectInIsolationAccTest
  - CreateWeavedProxiesOfExistingMocksAccTest

5.1.2 Tissage dans des classes générées

Tel qu'expliqué à la section 5.1.1, les Mocks virtuels ne peuvent être tissés que si ce tissage se déroule pendant l'exécution. Cependant, nous avons également établi que le LTW d'AspectJ ne peut être utilisé directement car il ne permet pas de contrôler quels aspects tisser, dans quels Mocks et quand procéder au tissage.

Pour ce faire, nous avons considéré plusieurs approches :

- créer un chargeur de classes (Class Loader) ;
- utiliser la configuration du tisseur dans le fichier « aop.xml » supporté en LTW ;
- tisser le Mock lors de sa création et avant son chargement grâce à l'utilisation de Proxies tissés après avoir appliqué un tisseur appelé via une extension [5] de l'API d'AspectJ.

4. En fait, il est possible de configurer le tisseur à l'aide du fichier « aop.xml » mais cette configuration n'est pas modifiable en cours d'éxécution à partir de Java.
5. L'extension utilise uniquement les mécanismes d'héritage et ne requiert pas de modifier AspectJ lui-même.

Les deux premières avenues se sont avérées impossibles à réaliser ou trop complexes.

La solution retenue consiste à tisser l'aspect dans la classe de Mock lors de la génération du Bytecode. Pour ce faire, nous devrons avoir accès au Bytecode du Mock *avant que ce dernier ne soit chargé* par le chargeur de classes. Ceci n'est possible que si nous pouvons nous « brancher » dans le cadre applicatif utilisé pour créer le Mock.

Cependant, afin d'offrir une solution la plus portable [fSI01] que possible, nous proposons une implémentation pouvant fonctionner avec presque la totalité des cadres applicatifs de création de Mocks. Cette solution est générique grâce à l'utilisation d'un Proxy [GHJV94] tissé qui est placé devant le Mock réel. Cette approche reste simple et peu visible par l'utilisateur.

De plus, Mock4Aj est conçu de manière à supporter une éventuelle intégration directe avec certains cadres applicatifs répandus. Cette extensibilité [fSI01] a déjà été prévue de manière à supporter l'ajout de nouveaux cadres sans requérir la modification du coeur de Mock4Aj.

La section 5.2.2 exposera les détails techniques concernant le tissage alors que la section 5.2.3 expliquera comment ces Proxies sont générés.

Note 4

L'objectif de Mock4Aj n'est pas de remplacer les cadres applicatifs de Mocks. Mock4Aj ne permet pas la création de Mocks mais permet le *tissage d'aspects* dans ceux-ci ainsi que leur configuration de manière à reproduire des *contextes* permettant les *tests d'aspects*.

Références techniques:

- *Scénarios utilisateurs liés :* M4AJ-01c (voir annexe A).
- *Test d'acceptations associés :*
  - WeaveWithAspectJAtRuntimeTest

5.1.3 Simulation de contextes

Afin de tester correctement la majorité des aspects, créer des Mocks de classes *existantes* n'est pas suffisant. En effet, AspectJ permet de définir des points de coupure restreignant le contexte d'exécution ou d'appel. De plus, ces contextes sont souvent définis à l'aide de patrons (wildcards) qui ne ciblent pas directement une classe existante du système.

Pour tester ces points de coupure, il est impératif d'offrir un mécanisme permettant la définition dynamique d'un contexte d'exécution ou d'appel.

Exécution d'une méthode

Considérant le point de coupure suivant :

```
execution( void foo() ) && within(*..T*)
```

Pour cet exemple, il faudrait idéalement essayer d'exécuter une méthode « foo » dans différentes classes qui doivent apparier ou non :

`Target.foo()`	`T.foo()`	`Sarget.foo()`
Oui	Oui	Non

Dans ce cas, une solution simple consiste à créer de « fausses » classes dans le test (Inner Class ou autre), de tisser l'aspect et de vérifier si ce dernier a apparié ou non :

```
public class SomeTest {
  public interface Target {
    void foo();
  }
  public interface T { ... }
  public interface Sarget { ... }

  @Test public void test() {
     // Ceci est une illustration et ne peut fonctionner telle quelle.
     Target targetMock = weavedMock(Target.class);
     T tMock = weavedMock(T.class);
     Sarget sargetMock = weavedMock(Sarget.class);

     assertWorkFor(targetMock, tMock);
     assertNotWorkFor(sargetMock);
  }
```

En plus, d'être complexe et peu élégante (pollution du test), cette façon de faire ne peut fonctionner dans tous les cas.

Par exemple, considérons la variante suivante du point de coupure qui restreint désormais les points de jonction en fonction du package dans lequel la classe est placée :

```
execution( void foo() ) && within(ca.ulaval.some..pkg.T*)
```

Dans ce cas, il n'est pas possible de créer une classe ou une interface à l'intérieur du test car le package ne correspondra pas. Une solution serait de créer ce package dans le Classpath Java et d'y créer les classes précédentes. Cependant, cette solution devient de plus en plus complexe, surtout quand le nom du package est lui-même un patron (ex. : `a..b`). Dans ce dernier cas, il faudrait alors créer plusieurs packages comme nous l'avons fait pour les noms de classes avec patrons.

Finalement, il existe aussi des cas *impossibles* à tester avec cette méthode. Afin d'illustrer ces cas, considérons le point de coupure suivant :

```
execution( void Target.f*() )
```

1

Dans ce dernier cas, la cible est précisée de manière exacte et précise. La méthode doit *absolument* être *directement dans* la classe « Target ». Elle ne peut donc pas être placée dans une classe héritant de « Target » ou encore une classe nommée « Target » mais placée dans un autre package ou encore comme classe embarquée (Inner Class).

De plus, il n'est pas possible d'utiliser un Mock de la classe « Target » car nous aurons besoin de créer des méthodes fictives qui ne sont pas réellement présentes dans cette classe :

`Target.foo()`	`Target.f() *`	`Target.too() *`
Oui	Oui	Non

* Méthode fictive n'existant pas réellement dans « Target »

Afin d'être en mesure de tester un tel point de coupure, il sera nécessaire de simuler une classe « Target » ayant les deux méthodes supplémentaires. Et en raison de la spécificité du point de coupure, cette classe de simulation devra avoir exactement le même nom et être dans le même package que la vraie classe « Target ». Or, cela est impossible puisque la classe existe déjà.

Le fait d'utiliser un Mock pourrait nous être utile ici car ce Mock est une copie « vidée » de la classe « Target ». Ce Mock simule le nom et est virtuellement dans le même package.[6] Il subsiste cependant deux problèmes liés à la façon dont fonctionne la majorité des cadres applicatifs de Mocks :

- ils créent les Mocks en générant une classe portant le même nom mais *héritant* de la classe doublée (le point de coupure tel qu'écrit exclut les classes enfants) ;
- ils ne permettent *pas d'ajouter des méthodes* qui n'existent pas dans la classe doublée.

Pour remédier à cette situation, il faudra donc offrir dans Mock4Aj la possibilité de créer un contexte d'exécution. Nous pourrons alors récréer les conditions exactes demandées par le point de coupure. Dans notre dernier exemple, il faudra alors créer une classe :

- avec le même nom « Target » ;
- être dans le même package ;
- ayant directement les méthodes (pas d'héritage) ;
- permettant l'ajout de méthodes fictives.

6. Cela est possible grâce à une astuce en Java qui consiste à suffixer le nom de la classe générée avec « $$suffixe » (consultez la section 5.2 pour plus de détails). AspectJ effectuera l'appariement sur ce qui précède le « $$ ».

Références techniques:

- *Scénarios utilisateurs liés :* [M4AJ-04b] (Voir annexe A)
- *Test d'acceptations associés :*
  - VerifyBehaviorOnPossibleMethodExecutionUsingInterfacesAccTest

Appel d'une méthode

Dans le cas de la primitive « call » d'AspectJ, elle permet, au même titre que « execution », de limiter sa portée avec d'autres primitives telles que « within », « this », « target », etc .

Par conséquent, les conclusions concernant l'exécution d'une méthode s'appliquent similairement au cas des appels :

```
call( void Target.f*() )
```
1

Cependant, la primitive « call » est plus complexe car l'origine de l'appel est différente de la cible :

```
call( void Target.f*() ) && within(O*) && this(Origin)
```
1

De plus, avec « call », l'interception (le point de jonction) se situe dans l'appelant (origine de l'appel) et non pas dans la classe appelée (la cible). C'est donc la classe appelante qui doit être tissée avec l'aspect et non la cible définie.

Ceci pose un problème beaucoup plus complexe qu'avec la primitive « execution » car il ne suffit pas d'appeler la méthode ciblée sur le Mock tissé pour déclencher l'action comme le montre la figure 5.1. Avec une primitive « call », il faut d'abord *invoquer une méthode qui, elle, appellera la cible.*

En effet, dans le cas d'un appel, c'est l'appelant qui doit être tissé (voir figure 5.1) et non pas la cible comme c'est le cas avec l'exécution. Intuitivement, la source (appel vers la cible) devrait être le test lui-même. Or, ce test est une classe qui est forcément déjà chargée en mémoire. Par conséquent, il n'est pas possible de la tisser à l'exécution avec uniquement l'aspect testé (voir section 5.1.1).

Il faut donc créer un objet dynamiquement généré qui fera l'appel comme le montre la figure 5.2. Cet objet étant généré à l'exécution lors du test, il pourra être tissé avec l'aspect testé avant son chargement par le chargeur de classes de Java (Java Class Loader).

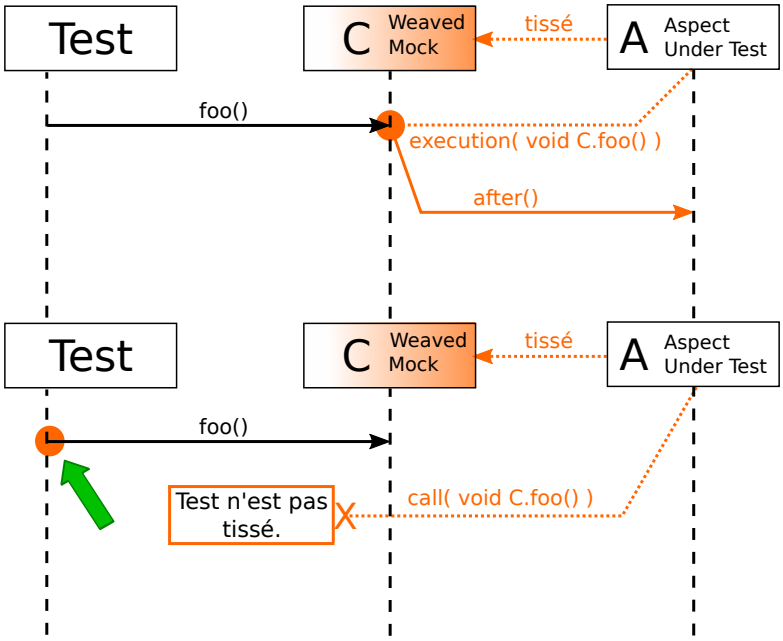


FIGURE 5.1: Diagramme de séquence démontrant l'impossibilité de tester une primitive « call » uniquement avec un Mock tissé contrairement à la primitive « execution ».

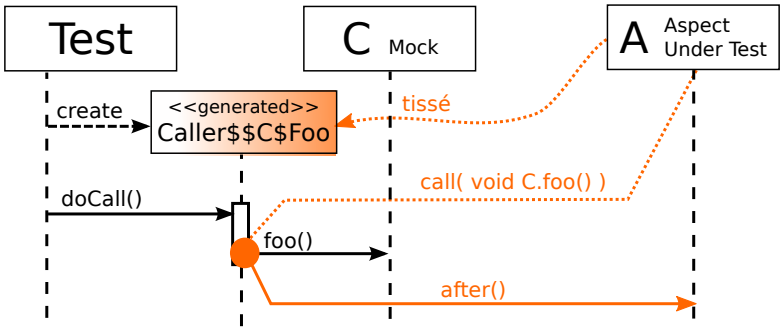


FIGURE 5.2: Diagramme de séquence montrant l'utilisation d'un « MethodCaller » pour tester une primitive « call ».

En résumé, il faut créer un objet sur mesure qui effectuera l'appel (un MethodCaller) en considérant les contraintes suivantes :

- il devra recréer le contexte d'appel désiré ;
- il ne pourra utiliser un Proxy car l'appel doit figurer directement dans le Bytecode tissé sans passer par un intermédiaire (voir section 5.2.4) ce qui requiert la génération directe à la volée de Bytecode par Mock4Aj ;
- il devra offrir une interface simple d'utilisation et qui cache la complexité de création et la génération du Bytecode (notamment permettre la sélection de la cible et de la source sans requérir l'utilisation de chaînes de caractères).

Ces contraintes sont loin d'être triviales et plusieurs difficultés sont à anticiper. Il faudra concevoir :

- des générateurs pour forger sur mesure des objets appelants ;
- générer des sélecteurs qui permettent de sélectionner la source et la cible sans exécuter réellement ces dernières.

Cela est particulièrement complexe dans la mesure où nous travaillons avec un langage statiquement typé et que, par conséquent, il faut toujours s'assurer que ces générateurs retournent des objets consistants et mimant les types réels qui permettent l'auto-complétion, le réusinage tout en ne demandant pas de forcer le typage (cast). La complexité doit également être cachée à l'utilisateur en lui offrant une syntaxe aisément utilisable.

La solution et les astuces techniques permettant d'arriver à une solution d'apparence simple pour l'utilisateur seront présentées à la section 5.2.4.

Incapacité d'utiliser un Mock comme source. À première vue, il peut sembler qu'utiliser un Mock tissé comme source serait une solution beaucoup plus simple ; à l'instar de ce que nous avons fait pour les cibles lors d'une exécution. Cependant cette solution n'est pas possible comme l'illustre la figure 5.3.

Ceci est lié à la technique utilisée (voir section 5.2.3) pour simuler le tissage d'un Mock. Pour y parvenir et afin d'être indépendant du cadre applicatif de Mock, nous utilisons un Proxy placé devant le Mock réel. Or, ce Proxy utilise un mécanisme de configuration asynchrone (callback) pour rediriger l'appel vers la cible du Proxy comme le montre la figure 5.3. L'introduction de cet intercepteur (tel que nommé avec CGlib), fait en sorte que l'appel ne se trouve plus réellement dans la classe tissée.

Nous devons donc absolument créer une classe sur mesure qui contiendra un appel réel (tel que présenté précédemment et illustré par la figure 5.2). Cela implique, par conséquent, de générer directement le Bytecode sans utiliser de Proxy. Nous présenterons cette solution en profondeur à la section 5.2.3.

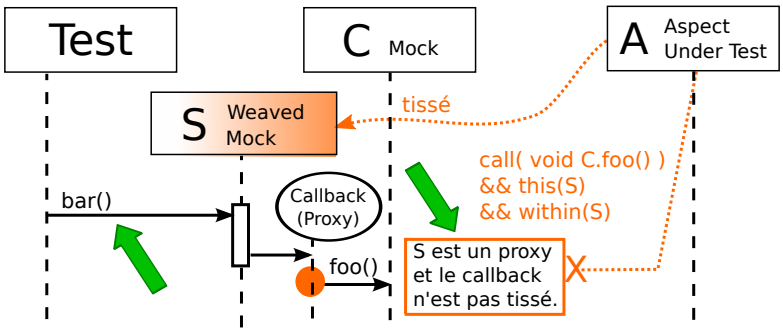


FIGURE 5.3: Diagramme de séquence démontrant l'impossibilité d'utiliser un Mock comme source.

Finalement, même si cette solution était techniquement possible, elle ne serait pas très intuitive pour l'utilisateur car cela supposerait d'appeler une méthode sur le Mock source qui contiendrait l'appel à apparier. Or, il n'est pas intuitif pour l'utilisateur d'appeler une méthode « `S.bar()` » alors que l'aspect testé ne se préoccupe que de « `C.foo()` ».

De plus, dans le cas d'un point de coupure plus générique (ex. : « `within(S*)` ») il faudrait créer un Mock d'une classe précise puisqu'un Mock est une doublure d'une classe réelle. Il s'imposerait alors de créer plusieurs classes pour les différentes variantes du patron, puis créer des Mocks pour chacune d'elles.

Références techniques:

- *Scénarios utilisateurs liés :* M4AJ-05a, M4AJ-06 (voir annexe A).
- *Test d'acceptations associés :*
  - SimulateCallsComingFromAFictitiousClassAccTest
  - SimulateCallsToATargetAccTest
  - SimulateCallsComingFromAnExistingClassAccTest

5.1.4 Support du réusinage et des IDEs

Nous avons déjà établi à la section 2.6.1 que le support au développement offert par les IDEs tiennent une place importante dans la trousse des développeurs Agile modernes. Il s'avérait donc primordial que Mock4Aj rende possible la prise en charge du réusinage et de l'auto-complétion.

Étant donné l'importance du point de vue de l'utilisabilité, un effort important a été mis en place afin de rendre Mock4Aj compatible malgré la complexité requise pour y parvenir.

Généralement, la prise en charge est automatique pour tout code Java et les cadres applicatifs n'ont rien de particulier à faire. Cependant, puisque Mock4Aj utilise de la réflexivité [Ora] et génère des classes dynamiquement, ce support n'est pas trivial en raison des classes générées qui n'existent pas au moment de la compilation. Par conséquent, l'IDE et ses outils statiques ne peuvent voir ces classes, les afficher et les considérer lors des réusinages.

Par exemple, il aurait été beaucoup plus simple de spécifier le nom des méthodes ou des classes à simuler à l'aide de chaînes de caractères. Or, avec cette façon de faire, si la méthode d'origine est renommée, la chaîne de caractère, elle, ne sera pas mise à jour par l'IDE comme le montre l'exemple 9.

Pour y parvenir il aura été nécessaire de mettre un important dispositif de simulation permettant la sélection de méthodes qui implique, encore une fois, la génération de classes dynamiques et d'intercepteurs comme nous le verrons à la section 5.2.

Exemple 9

Dans cet exemple, le « `Caller` » appellera la méthode « `getTime()` » sur un objet de type « `Date` ».

```
Date aDate = new Date();                                          1
MethodCaller caller = createCaller(aDate, "getTime");             2
long result = (Long) caller.doCall();                             3
```

– versus –

```
Date aDate = new Date();                                          1
long result = call(aDate).getTime();                              2
```

Voici les améliorations offertes par la deuxième version :

- la méthode appelée (`getTime`) n'est pas une chaîne de caractère ;
- si la méthode « `getTime` » est renommée dans la classe « `Date` », le test sera mis à jour automatiquement par l'IDE ;
- l'auto-complétion (ex. : `call(aDate).[COMPLÉTION]`) affichera la méthode « `getTime` » ;
- il n'est plus nécessaire de forcer le type (cast) du résultat car Java sait que « `getTime` » retourne un « long » alors que dans le cas d'une chaîne, Java ne sait même pas qu'il s'agit d'un nom de méthode.

5.1.5 Utilisation de l'API avec Java et d'autres technologies.

Étant donné que Mock4Aj est un cadre applicatif de test, il est nécessaire qu'il puisse utiliser conjointement d'autres technologies sans interférer.

De plus, le code testé peut lui-même utiliser divers cadres applicatifs et différentes technologies offertes en Java (JUnit, Mockito, JMock, Spring-test, ...). Dans certains cas, les tests doivent s'adapter à des technologies parfois très complexes.

Par exemple, une application web en Java peut être empaquetée dans un WAR qui sera ensuite déployé sur un serveur et, finalement, exécuté dans un JEE Container. Puisque ce JEE Container est un processus pouvant être un processus déjà démarré et indépendant, nous n'avons pas le plein contrôle sur l'application et, particulièrement, sur le chargeur de classes (Class Loader).

Avec ce type d'application, il devient complexe d'utiliser des Java Agents car ces derniers doivent être configurés au lancement de la JVM. Or, quand le WAR est déployé, la JVM est normalement déjà démarrée et le JEE Container s'exécute déjà. Pour pallier à cela, il existe diverses solutions qui peuvent varier d'un JEE Container à un autre ou qui impliquent de pré-configurer ce dernier.

Puisque nous développons un cadre applicatif de test, il est grandement préférable de ne pas requérir ce type de configuration complexe qui pourra interférer avec la manière de fonctionner de l'application testée ainsi qu'en fonction des technologies employées dans chacun des projets. Nous cherchons donc à offrir une solution *purement Java* et sans *Java Agent*.

Notre cadre doit donc offrir un API Java standard qui s'utilise à l'intérieur des méthodes de test JUnit classiques. Ceci incluant le démarrage et le tissage avec AspectJ.

5.2 Détails techniques de la solution

5.2.1 Survol des composantes

La figure 5.4 montre les grandes composantes et modules de Mock4Aj. La première chose à remarquer est la division nette entre l'API et les implémentations.

L'API ne fournit aucune implémentation (le « comment ») mais offrent des interfaces (abstractions) génériques (le « quoi ») que les clients peuvent utiliser comme service afin de tester un aspect.

Cette façon de faire, reconnue en génie logiciel, offre plusieurs avantages dont :

- *réutilisabilité* : facilite la réutilisation des composantes ;
- *extensibilité* : permet d'ajouter de nouvelles implémentations ou de spécialiser celles existantes sans avoir à modifier le cadre applicatif ;
- *choix de l'implémentation* : permet de choisir, voire combiner, les implémentations à utiliser ;

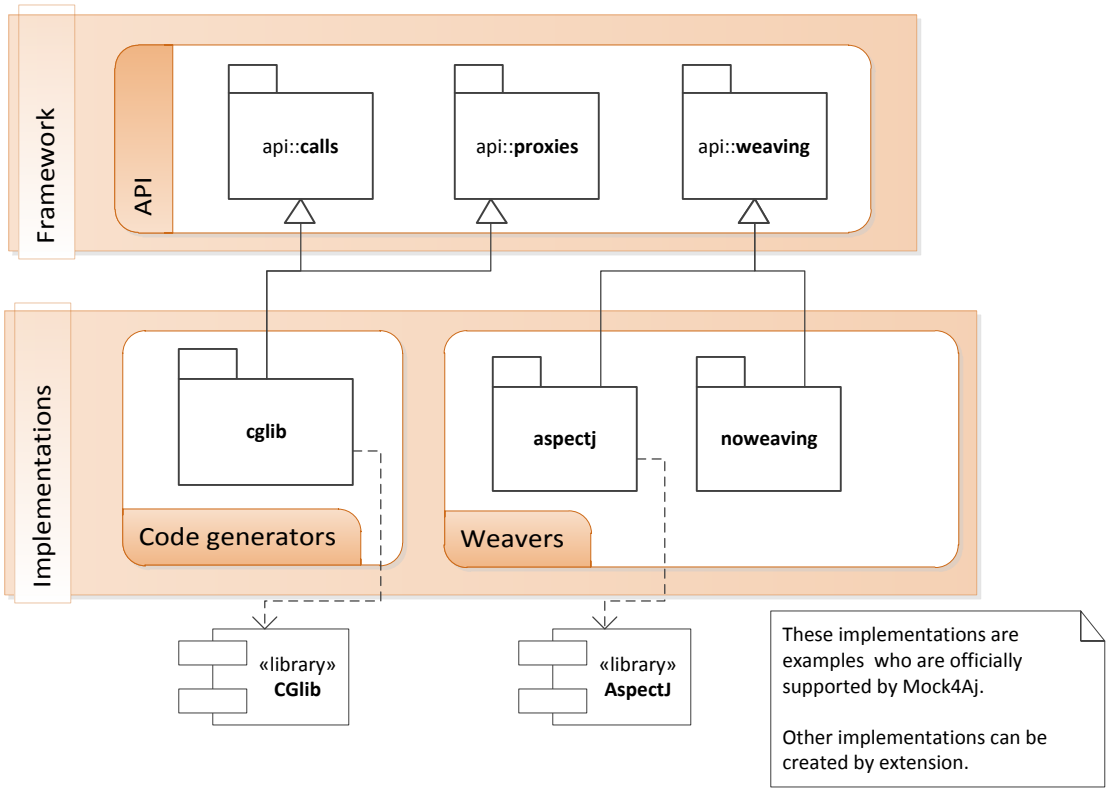


FIGURE 5.4: Diagramme des composantes principales de Mock4Aj.

Ainsi, il est facile d'aisément étendre (ajouter ou spécialiser) le cadre applicatif sans avoir à le modifier. En effet, le cadre a été conçu de manière à répondre, le plus possible [7], au principe du *Open-Closed Principle (OCP* [8]*)* [Mar96b] qui stipule qu'un module doit être ouvert aux extensions mais fermé aux modifications.

De plus, le cadre applicatif pouvant être utilisé par une foule d'applications et de personnes, il est important que son interface soit stable afin d'éviter des incompatibilités lors de changements. En fait, plus un module est utilisé (dépendances vers ce module), moins il ne peut changer sans répercutions fâcheuses. À plus forte raison quand il s'agit d'une API d'un cadre applicatif utilisé par des applications tierces. Ce principe et la métrique associée sont connus sous le nom d'*instabilité-abstraction* ou encore *The Stable Abstractions Principle (SAP)* [9] [Mar97, Mar02a].

7. La façade statique offrant les services ne répond pas à cette règle pour l'instant pour des raisons de contraintes techniques. Cependant, son utilisation n'est pas requise pour utiliser le cadre applicatif et n'est qu'une très mince couche offrant un sucre syntaxique. Il serait possible de corriger ce problème à l'aide d'un mécanisme d'enregistrement de plugiciels, par exemple.

8. Ce principe fait partie des principes S.O.L.I.D. reconnus en génie logiciel.

9. « Packages that are maximally stable [harder to change] should be maximally abstract. Instable packages

Ainsi, les clients (utilisateurs du cadre applicatif dans un test) ne dépendent pas de détails techniques mais d'abstractions génériques indépendantes de la technologie sous-jacente. Cela couvre partiellement une forme du principe appelé *Dependency Inversion Principle (DIP)* [Mar96a].

API

Mock4Aj offre une API permettant le tissage de Mocks. Cette API est générique, fortement abstraite et n'est pas tributaire de technologies spécifiques.

Elle est divisée en trois modules principaux :

- proxies pour la génération de « Proxy de Mocks » tissés (voir section 5.2.3) ;
- calls pour la simulation d'appels de méthodes dont l'origine répond à un contexte donné et tissé (voir section 5.2.4) ;
- weaving pour le support du tissage dynamique des objets (voir section 5.2.2).

Implémentations

Actuellement, Mock4Aj fournit certaines implémentations permettant d'offrir les services offerts par l'API :

- CGlib comme générateur de Bytecode à la fois pour la simulation des appels que pour la création de Proxies ;
- AspectJ comme tisseur d'aspects par défaut.

Il serait possible de remplacer ces implémentations officielles par d'autres sans problème, simplement en implémentant les interfaces et classes abstraites de l'API.

L'architecture est également élaborée de manière à permettre la combinaison de différentes implémentations dans chacune des catégories.

Exemple 10

À titre d'exemple, il serait théoriquement possible de créer une nouvelle implémentation utilisant BCEL pour créer des Proxies.

On pourrait ensuite utiliser ces implémentations en combinaison comme suit :

- AspectJ comme tisseur ;
- BCEL comme générateur de code afin de créer les Proxies ;
- CGlib comme générateur de code afin de simuler les appels.

should be concrete. The abstraction of a package should be in proportion to its stability [Mar97] ».

5.2.2 Tissage d'objets dynamiques

Le premier problème à résoudre consiste à parvenir à contrôler le tissage d'aspects dans des classes dynamiquement générées. Sans cette composante, le cadre applicatif n'a aucune raison d'être et n'aurait aucune valeur.

Il convient donc en premier lieu de s'attaquer à l'élément ayant le plus de valeur mais également celui qui est le plus risqué.

L'objectif est d'avoir un module permettant :

- de tisser uniquement les aspects sélectionnés pour chaque demande de tissage ;
- de les tisser dans des classes dynamiquement générées [10].

En effet, sans être en mesure de tisser des classes sur demande avec un sous-ensemble d'aspects, générer des Proxies et simuler des appels ne servirait à rien.

API offert

D'un point de vue du client, une interface (classe abstraite) Weaver est offerte et définit les services d'un tisseur.

En résumé, un tisseur peut :

- enregistrer des aspects à tisser ;
- désenregistrer des aspects ;
- tisser les aspects enregistrés dans une classe à partir de son Bytecode (`bytecodeOrginal[] + nomDeClasse→bytecodeTisse[]`).

Implémentation utilisant AspectJ

Par défaut, Mock4Aj fournit une implémentation de ce tisseur qui utilise AspectJ. Nous avons déjà établi à la section 5.1.1 qu'il n'est pas possible d'utiliser directement le compilateur (CTW) ou le tisseur dynamique (LTW) d'AspectJ.

Avant de détailler la solution, expliquons brièvement comment fonctionne AspectJ en LTW.

AspectJ en LTW avec un agent Java. Normalement, AspectJ peut tisser des classes lors de leur chargement par le chargeur de classe (Java Classloader). Pour ce faire, il utilise un mécanisme prévu à cet effet dans le standard Java que l'on nomme un agent Java (Java Agent).

Il s'agit d'un mécanisme permettant d'enregistrer un module qui sera automatiquement appelé lorsqu'une nouvelle classe est chargée par le chargeur. À ce moment l'agent a la possibilité de

10. Par classe dynamiquement générée, nous entendons une classe pour laquelle il n'existe pas de code source Java (.java) statique et dont le contenu est généré dynamiquement lors de l'exécution. Il s'agit, grosso modo, d'écrire du code Java qui génère une nouvelle classe qui sera chargée par le chargeur de classe.

faire des transformations et de manipuler le Bytecode (le source binaire) de la classe chargée avant que celle-ci ne soit définitivement chargée en mémoire et disponible pour utilisation.

Notons qu'un tel agent doit être enregistré avant le démarrage de la JVM (la machine virtuelle Java qui interprète le Bytecode). Une fois le programme lancé [11], Java n'offre pas de méthode standardisée [12] pour ajouter un nouvel agent ou pour accéder à ceux enregistrés.

De son coté, l'agent d'AspectJ intercepte toutes les nouvelles classes chargées par le chargeur et fait une tentative de tissage (si un point de coupure apparie). *Il fait cela pour toutes les classes chargées et avec tous les aspects enregistrés ou trouvés au démarrage.* C'est l'une des principales raisons pour lesquelles cela ne peut pas fonctionner directement dans notre cas.

Par contre, l'agent d'AspectJ (org.aspectj.weaver.loadtime.Agent) n'effectue pas lui-même le tissage. Pour ce faire, il utilise un WeavingAdapter comme point d'entrée dans l'API de tissage d'aspectJ (aspectjweaver.jar).

Tisseur dynamique de Mock4Aj avec AspectJ. Dans notre cas, nous voulons créer une implémentation de Weaver qui utilise AspectJ afin de réaliser le tissage (la modification du Bytecode insérant les aspects).

De plus, contrairement à une utilisation normale d'AspectJ dans un programme en exécution, nous ne voulons pas tisser toutes les classes avec tous les aspects. Nous voulons contrôler le *moment*, les *aspects à tisser* ainsi que les *classes dans lesquelles les tisser*.

Pour ce faire, nous pouvons utiliser la même méthode que l'agent d'AspectJ et utiliser l'API de tissage via un WeavingAdapter.

Cependant, le WeavingAdapter par défaut, ne répond pas complètement à nos besoins pour diverses raisons dont le fait qu'il n'est pas possible d'y désenregistrer des aspects et de le réinitialiser. De plus, cet adapteur lit les configurations d'AspectJ (aop.xml) alors que nous désirons contrôler complètement le processus à partir du code du test.

Nous avons donc créé une extension DynamicRuntimeWeavingAdaptor qui hérite du WeavingAdapter d'AspectJ. Cette extension permet de contrôler complètement le tissage via un API Java à partir du code lors de l'exécution.

Cela requiert de copier certaines parties et d'utiliser certaines astuces car AspectJ n'est pas pleinement extensible (principe OCP).

11. Une fois la JVM lancée pour être précis. Cependant, assumons pour la suite, qu'un seul programme est lancé par JVM.

12. Certaines implémentations de JVM peuvent fournir ce service mais ce n'est pas normalisé dans un JSR (standard normalisé). La mécanique varie donc d'une JVM à une autre et n'offre aucune garantie de support à long terme.

Il a également été nécessaire de créer un nouveau type de « monde » (World [13]) utilisant la réflexivité de Java afin de trouver les types à résoudre. En effet, puisque que AspectJ n'est plus un agent Java, il ne voit pas toutes les classes qui sont chargées par le chargeur, ce qui se traduit par un monde incomplet. Notre monde adapté permet, de manière simplifiée, d'extraire les informations sur une classe à partir du « monde » de Java si l'information n'est pas disponible dans le « monde » d'AspectJ.

Résumé. L'avantage principal de cette technique est qu'elle *ne requiert aucune modification au code d'AspectJ* qui est simplement utilisé comme bibliothèque (Library/JAR).

De plus, elle nous offre la possibilité de contrôler entièrement le tissage *à partir d'un code purement Java* et à partir d'un *chemin d'exécution prévu par AspectJ* et non pas à partir d'astuces instables.

5.2.3 Proxy

La deuxième étape consiste à réaliser un module permettant de tisser des Mocks. L'obtention de Mocks tissés nous permettra de les utiliser comme stimulus afin de déclencher des actions sur l'aspect testé [14].

Pourquoi utiliser des Proxies

À première vue, le plus simple serait d'utiliser notre tisseur (voir section 5.2.2) afin de tisser un Mock virtuel existant.

Cependant cela n'est pas possible directement car il n'est pas possible de tisser une classe Java qui a déjà été chargée par le chargeur de classe. Plus précisément, il n'est pas aisément possible non plus d'obtenir son Bytecode une fois chargé afin d'en créer une nouvelle copie modifiée (tissée).

Il faudrait donc intercepter le Bytecode avant le chargement. La manière la plus simple d'y parvenir serait d'utiliser un agent Java [15]. Or, nous avons déjà rejeté cette solution à la section 5.2.2.

L'autre solution serait d'utiliser notre tisseur afin de tisser un Mock virtuel pendant sa création.

13. Le monde contient, entre autre, l'ensemble des classes et aspects connus pas AspectJ. AspectJ se sert de son monde afin de résoudre les dépendances et trouver les types. Normalement, une classe doit être dans le monde pour qu'un point de coupure puisse l'apparier.

14. Rappelons que les greffons d'un aspect sont exécutés lorsqu'un point de coupure lié de l'aspect apparie un point de jonction correspondant dans une classe tissée. Ainsi, un comportement dans un Mock tissé nous permettra de vérifier que l'aspect a apparié et que le bon greffon s'est exécuté. Voir section 4.2.3.

15. Il serait également possible d'utiliser un chargeur de classes personnalisé. Cette solution a été expérimentée avec un prototype mais ne peut fonctionner pleinement dans notre cas pour diverses raisons liées à la manière dont les chargeurs fonctionnent.

Étant donné que la majorité des Mocks sont des Mocks virtuels créés par des cadres applicatifs spécialisés dans cela (ex. : Mockito), il faudrait alors un moyen d'insérer le tisseur dans le processus de fabrication utilisé par ces cadres applicatifs externes.

Même si cela est possible en fonction du cadre utilisé [16], cette solution serait tributaire d'un cadre applicatif de Mock. Or, nous avons déjà établi au chapitre 4 que nous désirons nous intégrer de manière la plus transparente possible aux outils déjà utilisés par les entreprises. Nous voulons, par conséquent, *offrir une solution qui peut fonctionner avec le maximum d'outils de Mock déjà employés dans l'industrie.*

Par contre Mock4Aj a été conçu de manière à supporter des connecteurs (bindings) avec différents cadres applicatifs courants. Bien qu'aucun de ses connecteurs ne soit actuellement disponible ce support pourrait être ajouté prochainement.

L'avantage de tels connecteurs serait de simplifier le nombre d'étapes requises par l'utilisateur. En plus, cela serait encore plus intuitif car, après tout, ce que l'utilisateur veut obtenir est un Mock tissé. Alors qu'avec un Proxy, on lui donne un Proxy tissé du Mock qu'il doit d'abord créer. Cela requiert à l'utilisateur de comprendre un minimum du fonctionnement de Mock4Aj et il serait préférable en terme d'utilisabilité que ça ne soit pas le cas.

Fonctionnement des Proxies

Un Proxy [GHJV94] est « un substitut ou un élément de substitution (placeholder) pour un autre objet afin d'en contrôler l'accès [GHJV94] ».

Dans notre cas, nous utilisons un Proxy que nous plaçons devant un Mock que nous désirons tisser.

La figure 5.5 montre le fonctionnement du Proxy qui a les caractéristiques suivantes :

- Le Proxy est du même type et expose donc les mêmes signatures (méthodes) que l'objet d'origine (type du Mock) ;
- Le Proxy est configuré de telle sorte que lorsqu'une méthode est appelée sur le Proxy, l'appel est « relancé » vers le Mock d'origine ce qui fait que le Mock pourra faire son travail de Mock normalement ;
- le Proxy sera tissé ce qui permettra à l'aspect de s'exécuter ou non selon le cas.

Étant donné que le Proxy est créé par une implémentation de Mock4Aj, il nous est possible de le tisser lors de sa création et avant son chargement. Ce qui nous était impossible avec un Mock existant et donc déjà chargé.

Une fois le Proxy obtenu, il sera possible de l'utiliser comme stimulus pour l'aspect tissé. De plus, les actions de l'aspect sur le Proxy seront automatiquement reproduites sur le Mock réel.

16. Mockito ne permet pas cela sans avoir à en modifier le code source.

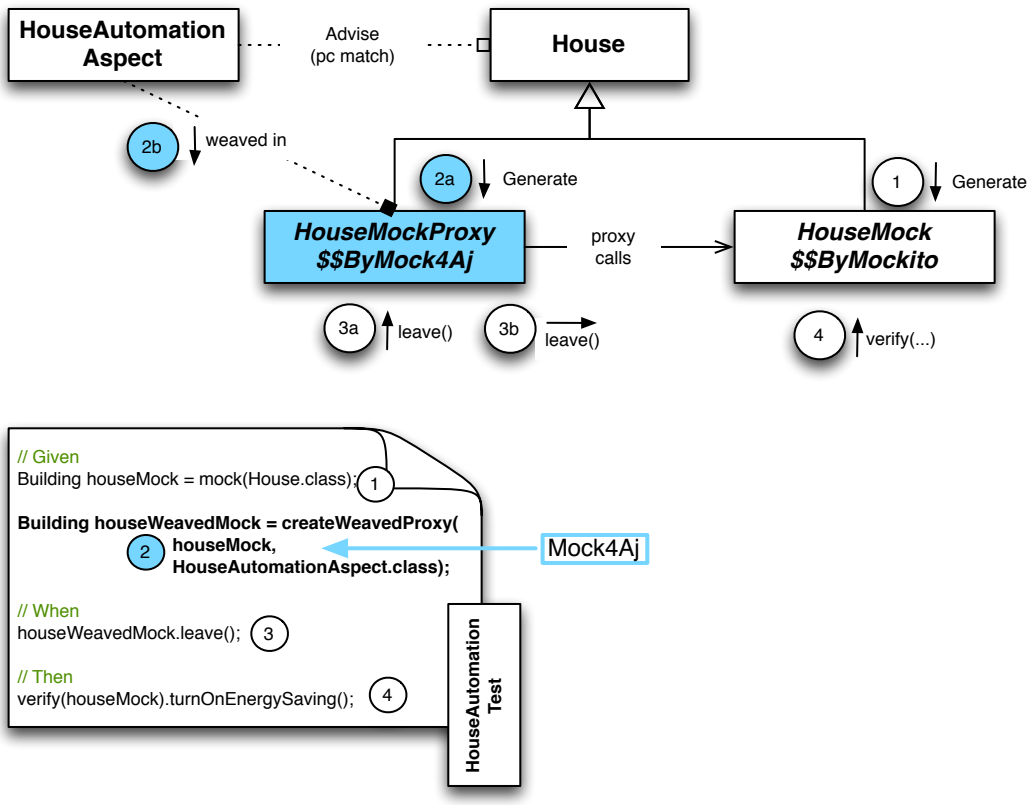


FIGURE 5.5: Fonctionnement d'un Proxy tissé d'un Mock.

Ce faisant, il sera possible de vérifier les comportements attendus sur le Mock comme avec tous Mocks réels.

Un défaut de cette approche concerne l'utilisabilité. En effet, l'utilisation de Mock4Aj doit comprendre qu'il doit stimuler l'aspect via le Proxy mais doit vérifier le résultat sur le Mock réel. Il ne peut faire ses vérifications directement sur le Proxy comme cela semblerait intuitif.

Par contre, cette solution permet d'utiliser pratiquement n'importe quel cadre applicatif de Mocks virtuels. Nous pensons que, dans un premier temps, le fait de pouvoir continuer à utiliser son outillage existant [17] constitue un plus grand avantage dans la balance que la perte d'utilisabilité.

Cependant, tel qu'expliqué précédemment, Mock4Aj est conçu afin de permettre la création de connecteurs spécifiques éliminant ainsi ce problème.

17. Changer un outil utilisé dans pratiquement tous les tests d'un projet et avec lequel toute l'équipe est familière est potentiellement très coûteux en temps mais aussi en apprentissage.

API

Mock4Aj offre une interface très simple pour la création de Proxies via WeavedProxyFactory qui permet la création d'un Proxy d'un Mock tissé à l'aide d'un tisseur (voir section 5.2.2) donné :

```
<T> T createWeavedProxy(final T objectToProxy,
                        final Weaver weaver);
```

Notons qu'il est théoriquement possible de créer un Proxy tissé de n'importe quel objet. Il n'est pas nécessaire qu'il s'agisse d'un Mock. Cependant, créer un Proxy tissé d'un objet dont le type est une classe réelle (pas dynamiquement générée) constitue un danger car cet objet pourrait avoir déjà été tissé par le compilateur statique (ex. : AJC). Cela pourrait alors causer des résultats de tests erronés ou encore boucler infiniment.

Mock4Aj devrait normalement essayer de détecter ce cas et informer l'utilisateur via un avertissement. Mock4Aj a cependant comme but explicite de tester les aspects à l'aide de Mocks. Il ne s'agit donc pas d'une limitation.

Implémentation avec CGlib

Nous offrons, par défaut, une implémentation (CglibWeavedProxyFactory) utilisant CGlib comme générateur de Bytecode.

Note 5

CGlib (Code Generation LIBrary) est une bibliothèque permettant la génération dynamique de classes à l'exécution. Il offre une couche de haut niveau par dessus ASM qui est le réel manipulateur de Bytecode.

L'utilisation première de CGlib est la création de classes personnalisées en permettant l'héritage et l'implémentation de classes et interfaces à l'exécution. Dans ce cas, le mécanisme le plus courant est l'utilisation de « Callbacks » afin de définir le comportement de la nouvelle classe créée.

Il est cependant possible d'en faire une utilisation avancée en définissant des classes complètement sur mesure en spécifiant le Bytecode à insérer dans les méthodes (comme nous le ferons à la section 5.2.4).

Description du Proxy créé. Afin de créer un objet Proxy « P » tissé d'un Mock « M », nous demandons à CGlib de générer une classe « C_P » qui :

- mime le type de « M »
  - implémente les même interfaces que « M » ;
  - hérite de la même classe concrète dont « M » est le Mock ;
- utilise notre stratégie de tissage (CglibWeavingGeneratorStrategy) qui est configurée avec un tisseur de Mock4Aj et qui transformera (tissage) le Bytecode avant son chargement ;
- utilise un intercepteur de méthode [18] (ProxyMethodCallback) qui aura pour effet de transférer tous les appels faits sur « P » à « M ».

Appariement de la classe générée. Malgré cela, un problème demeure, comment faire croire à AspectJ qu'il s'agit de la même classe afin qu'il apparie correctement.

Par exemple, le point de coupure suivant est très stricte sur la classe concernée :

```
execution(* info.rubico.Target.theMethod(..))
&& within(Target)
&& this(info.rubico.Target)
```

Dans cet exemple, la méthode interceptée doit être dans une classe nommée « `Target` » mais également spécifiquement dans cette classe « `Target` » existante. Or, notre Proxy n'est pas spécifiquement cette classe mais une autre classe générée qui mime « `Target` ».

L'astuce consiste à préfixer la classe générée par le nom ciblé puis de faire varier le nom en ajoutant un suffixe propre après les symboles « $$ ». Ainsi, le Proxy de la classe « `info.rubico.Target` » portera un nom débutant par
« `info.rubico.Target$$EnhancerProxyByMock4Aj$$cbd4461a` » [19].

Cette convention de nommage est utilisée par pratiquement tous les générateurs de code et le JLS (Java Language Specification) [SM00a] recommande cette convention : « The $ character should be used only in mechanically generated source code [...] [SM00b] ».

AspectJ n'utilisera que la première partie du nom afin de déterminer l'appariement des points de coupure. Ainsi pour AspectJ, une classe « X » est équivalente à « X$$... » concernant l'appariement.

Cette astuce pourrait ne pas fonctionner avec d'autres tisseurs. Cependant, cela serait surprenant car il s'agit d'une convention fortement répandue. De plus, puisque le tisseur génère

18. L'intercepteur de méthode est l'objet (« Callback ») qui sera appelé lorsqu'une méthode est appelée sur un objet créé par CGlib. C'est en quelque sorte le « contenu » à exécuter pour les méthodes créées. Cette technique permet d'écrire le code à exécuter en Java dans une classe compilée plutôt que d'écrire le Bytecode à insérer.
19. Le nom encode ainsi l'adresse mémoire de l'objet pour lequel on veut créer un Proxy ansi que le générateur utilisé.

normalement lui-même des classes dynamiquement, il devrait répondre aux mêmes conventions et ne considérer que le préfixe sans quoi il ne pourrait potentiellement correctement apparier ses propres classes générées.

Il s'agit cependant que d'une hypothèse à tester avec chacun des tisseurs. Notons que Mock4Aj utilise une stratégie de nommage (WeavedProxyNamingPolicy) afin de déterminer le nom à donner à la classe générée. Il serait donc relativement aisé de l'adapter ou d'en fournir une nouvelle répondant aux exigences de futurs tisseurs.

Résumé. Cette section a présentée une version résumée du fonctionnement des Proxies. Pour comprendre le fonctionnement détaillé ou en saisir les subtilités, consulter le code de Mock4Aj ainsi que sa documentation est recommandé.

Références techniques:

Nous n'avons volontairement pas couvert les différents cas possibles comme, par exemple, les différentes primitives supportées. Pour plus de détails concernant le comportement attendu pour différents cas, nous recommandons de consulter le test d'acception « CreateWeavedProxiesOfExistingMocksAccTest » qui décrit les possibilités à l'aide de scénarios (exemples) exécutables.

5.2.4 Simulation d'appels

L'objectif de la simulation d'appels est de créer une classe dynamiquement qui simule un appel vers une cible. La classe reproduit un contexte fictif préalablement configuré.

Les sections suivantes décrivent succinctement le fonctionnement interne mais qui est caché à l'utilisateur qui n'a pas besoin de connaître ces détails et astuces techniques pour utiliser Mock4Aj.

Afin de proposer une syntaxe simple à l'utilisateur et qui cache toute la complexité liée à la génération dynamique du code, nous devons d'abord avoir un mécanisme de sélection de la cible qui fonctionne avec l'auto-complétion et les outils de réusinage des IDE modernes.

La figure 5.6 montre l'ensemble des classes et du flot nécessaire afin de :

1. définir un contexte (ex. : source de l'appel) ;
2. obtenir un sélecteur qui permettra de sélectionner la méthode à appeler (cible) ;
3. simuler l'appel.

FIGURE 5.6: Fonctionnement du simulateur d'appels

Sélection de la méthode ciblée

Cette étape est nécessaire pour deux raisons :

- afin de permettre à l'utilisateur de sélectionner la méthode par une invocation plutôt que de fournir le nom sous la forme d'un texte ;
- pour faire en sorte que la méthode ne soit pas réellement appelée à partir du test.

Voici un exemple illustrant ces cas :

```
CAS 1: Texte
  call(contexte, objet, "methode", param1, ...);

CAS 2: Appel direct
  call(objet.methode(param1), contexte);

CAS 3: Avec sélection
  call(objet, contexte).methode(param1);
```

Le premier cas est difficile à lire et ne résiste pas au renommage de la méthode ou à un changement de signature. Ainsi, si la méthode est renommée ou des paramètres ajoutées, l'IDE n'a aucun moyen de savoir que la chaîne de caractères correspond dans les faits, à une méthode. De plus, la syntaxe est loin d'être intuitive et ne permet pas l'auto-complétion.

Le deuxième cas est le plus simple mais pose un problème technique car l'appel sera immédiatement fait à partir du test et non pas à partir de la source configurée dans le contexte. En effet, la méthode sera immédiatement appelée sur l'objet avant même l'appel vers « call ». Finalement, cette syntaxe poserait un problème dans le cas de méthode dont le retour est « void » car Java ne permet pas de passer un « void » comme paramètre à la méthode « call ».

Nous avons donc choisi une syntaxe qui fait en sorte de permettre la sélection via un faux appel de méthode.

Résumé du fonctionnement. Pour ce faire, la méthode désirée doit être appelée sur un objet qui mime le type de l'objet (la classe qui contient la méthode). L'astuce est de faire retourner par « call » une classe générée qui est du même type que « objet ». Sauf que cette classe de sélection est créée de manière à sélectionner la méthode appelée.

Pour ce faire, le simulateur procède comme suit :

1. Il utilise un générateur de sélecteurs (SelectorGenerator) qui permet de générer un sélecteur (Selector$$) qui mime le type de l'objet ;
2. Le simulateur configure le générateur avec un intercepteur (CallInterceptor) qui fonctionne comme une Callback. C'est cet intercepteur qui sera appelé quand l'utilisateur sélectionnera la méthode sur le sélecteur retourné (`Selector$$.methode()`). Dans les faits, le sélecteur agit comme un Proxy.

Simulation de l'appel

Une fois le sélecteur retourné par la méthode « call », l'utilisateur n'a qu'à appeler la méthode ciblée sur ce sélecteur. Ce faisant, il sélectionnera la méthode et déclenchera la simulation du même coup.

Pour mieux illustrer cela, séparons l'exemple en deux lignes :

```
[1] House houseSelector = call(houseMock, context);
[2] houseSelector.leave();
```

C'est la deuxième ligne qui lance véritablement la simulation puisque la première ne fait que retourner un sélecteur : une classe générée de manière à limiter le type House mais dont la méthode « leave » ne fait que lancer une simulation d'appel vers la vraie méthode mais à partir du contexte source configuré et non pas du test.

Résumé du fonctionnement. L'objectif de la simulation est de créer un contexte qui simule une source d'appel tissée par un aspect qui a un point de coupure « call ». Comme nous l'avons déjà expliqué, c'est la classe d'origine de l'appel qui doit être tissée et non la cible. De plus, il doit être possible de simuler un appel émanant d'une classe fictive afin de tester les différents patrons.

Pour ce faire, le simulateur procède comme suit :

1. L'appel vers la méthode « leave » du sélecteur agit comme proxy et transmet l'appel à l'intercepteur (CallInterceptor) qui a préalablement été configuré avec le contexte désiré.
2. L'intercepteur utilise un générateur d'appels (CallerGenerator) qui permet de générer une classe sur mesure (Caller$$) qui
   a) répond à tous les critères du contexte (nom de la classe d'origine, type, interfaces implémentées, etc.) ;
   b) a une méthode « doCall » qui a un appel direct vers la méthode ciblée (`houseMock.leave()`) ;
   c) est tissé avec les aspects à tester.
3. L'intercepteur appelle la méthode « doCall » sur « Caller$$ ».
4. La méthode « doCall » fait ensuite l'appel réel.

Cela a pour effet de produire l'appel réel sur le Mock à partir d'une classe tissée.

Classe Caller générée. Comme il est impératif que l'appel provienne directement de la classe simulant le contexte d'origine (Caller$$), il est nécessaire de générer directement le Bytecode sans utiliser un Proxy ou autre mécanisme indirect. En fait, les implémentations de CallerGenerator, sont des générateurs de Bytecode qui produiront une méthode « doCall » différente et sur mesure en fonction du contexte.

Voici un exemple du Bytecode généré :

```
// Compiled from <generated> (version 1.2 : 46.0, no super bit)
public class MySourceX$$CallerByMock4Aj$$Target_Enhanced$retInt$$58ba75e6
    implements info.rubico.mock4aj.testdata.ParentInterfaceSource,
               info.rubico.mock4aj.api.calls.MethodCaller {

  public java.lang.Object target;

  public MySourceX$$CallerByMock4Aj$$Target_Enhanced$retInt$$58ba75e6();
    0 aload_0 [this]
    1 invokespecial java.lang.Object() [15]
    4 return


  public void Mock4Aj$setTarget(java.lang.Object arg0);
    0 aload_0 [this]
    1 aload_1 [arg0]
    2 putfield MySourceX$$CallerByMock4Aj$$Target_Enhanced$retInt$$58ba75e6.target : java.lang.Object [19]
    5 return


  public java.lang.Object doCall(java.lang.Object[] arg0);
     0 aload_0 [this]
     1 getfield MySourceX$$CallerByMock4Aj$$Target_Enhanced$retInt$$58ba75e6.target : java.lang.Object [19]
     4 checkcast info.rubico.mock4aj.acceptance.testdata.Target$$EnhancerByMockito$$c317a4f5 [23]
     7 invokevirtual info.rubico.mock4aj.acceptance.testdata.Target$$EnhancerByMockito$$c317a4f5.retInt() : int [27]
    10 new java.lang.Integer [29]
    13 dup_x1
    14 swap
    15 invokespecial java.lang.Integer(int) [32]
    18 areturn
    19 new info.rubico.mock4aj.api.exceptions.EncapsulatedExceptionThrownByTarget [36]
    22 dup_x1
    23 swap
    24 invokespecial info.rubico.mock4aj.api.exceptions.EncapsulatedExceptionThrownByTarget(java.lang.Throwable) [39]
    27 athrow
      Exception Table:
        [pc: 7, pc: 10] -> 19 when : java.lang.Throwable
}
```

Noter que le mécanisme est en place afin de permettre la bonne gestion transparente des exceptions lancées par la méthode appelée ainsi que de la valeur de retour. Tout est conçu de manière à être le plus transparent possible pour l'utilisateur qui recevra la valeur de retour et les exceptions comme s'il avait appelé directement la méthode à partir du test.

Références techniques:

Pour comprendre les différentes possibilités et limitations concernant les contextes d'appels, consultez les tests d'acceptation suivants ainsi que le code source des classes concernées.

- SimulateCallsComingFromAFictitiousClassAccTest
- SimulateCallsComingFromAnExistingClassAccTest
- SimulateCallsToATargetAccTest

Chapitre 6

Conclusion

La programmation orientée aspect offre de nouvelles possibilités architecturales permettant de mieux regrouper les préoccupations transverses et réduire leur dispersion et l'enchevêtrement qu'elles causent.

Pour ce faire, nous avons d'abord procédé à une étude d'exploration afin d'identifier des problèmes potentiels. Ensuite, nous avons réalisé un cadre applicatif (framework) qui permet de tester unitairement des aspects.

Étude préalable. Notre objectif initial était de mieux comprendre les effets d'un tel paradigme dans le contexte d'un projet d'entreprise se déroulant en mode Agile. Pour ce faire, nous avons monté une étude préliminaire dont le but était de nous donner une idée des problèmes potentiels avec AspectJ et un projet utilisant Scrum et certaines pratiques d'eXtreme Programming.

Dans un premier temps, nous avons choisi de nous concentrer sur des développeurs n'ayant aucune connaissance de l'AOP et dont l'expérience en programmation objet est faible. Toutefois, l'étude a été réalisée dans un contexte de projet Agile (avec accompagnement) et avec des technologies d'envergure et qui sont couramment utilisées en entreprise (Maven, Java EE, web, etc.).

Plusieurs problèmes potentiels ont été soulevés [BL10] par notre groupe d'expérimentation. Mais le principal problème perçu était d'écrire des tests unitaires pour des aspects. Dans les faits, tous les répondants ont éprouvé des difficultés à écrire de tels tests.

Cette étude ayant comme objectif de sélectionner plus judicieusement un empêchement à mitiger, nous avons décidé de nous attaquer à ce problème en concevant un cadre applicatif (framework) permettant d'écrire des tests unitaires d'aspects.

Mock4Aj. Mock4Aj est le cadre applicatif que nous avons développé afin d'aider l'écriture de test unitaires d'aspects. C'est un cadre applicatif en Java qui permet d'isoler un aspect et

de simuler différents contextes afin de vérifier le comportement mais également l'appariement des points de coupures.

Notre objectif était de concevoir un cadre facile à utiliser (voir annexe **??**) pour un développeur habitué aux outils existants pour les tests unitaires en Java (sans AspectJ). Nous avons cherché à utiliser les mêmes techniques et pratiques actuelles et réputées pour la conception de tests unitaires en mode Agile. Ce faisant, nous voulions réduire la courbe d'apprentissage mais également éviter à un développeur de constamment avoir à changer sa façon de travailler selon qu'il s'agisse d'un aspect ou d'une classe.

Pour ce faire, nous avons caché toute la complexité liée à la génération de classes dynamiques derrière une interface simple et épurée. Nous avons porté une grande attention à la manière de sélectionner et définir un contexte qui se lit aisément et qui est très intuitive. Même si cette façon de faire ajoute de la complexité au cadre lui-même, la priorité était d'offrir un cadre applicatif le plus simple d'utilisation possible pour l'utilisateur. Nous avons utilisé une syntaxe qui permet d'expliciter l'intention du test avec une forme de chaînage qui peut se lire presque comme une phrase.

Nous voulions également un cadre applicatif qui aurait peu de chance d'entrer en conflit avec les outils et technologies utilisés dans l'industrie pour des projets Java par une équipe Agile. C'est pourquoi notre solution est écrite en « Java pur » et peut être compilée par n'importe quel compilateur Java et exécutée par une JVM standard. Aucun mécanisme externe ou pré-compilation n'est nécessaire.

Finalement, nous avons veillé à ce que la solution soit extensible en respectant les principes d'une bonne conception orientée objet et architecturale comme les principes S.O.L.I.D. [Mar02b] De plus, une attention particulière a été portée à la propreté du code [Mar08b] afin qu'il puisse être aisément maintenu.

En conclusion, nous avons conçu et rendu public un cadre applicatif qui permet de tester unitairement des aspects (en AspectJ) tout en respectant les règles de l'art en matière de tests unitaires en mode Agile.

Discussion et travaux futurs

L'étude réalisée en première partie était une étude exploratoire et n'avait pas comme objectif de déterminer précisément et avec certitude un palmarès des problèmes d'utilisation de l'AOP en mode Agile. Une telle étude contrôlée serait intéressante afin de déterminer sur un plus large échantillon les raisons de la faible adoption avec le temps de l'AOP.

Concernant Mock4Aj, le but était de développer une solution à la portée du monde industriel sans requérir d'importants changements de culture et de pratiques. Nous cherchions une solution qui serait facilement adoptable dans l'immédiat. Cependant, il ne s'agit peut-être pas de la meilleure façon théorique de tester unitairement des aspects. Ou encore peut-être faudrait-il

remettre en cause les règles de l'art concernant ce type de test en mode Agile quand il est question d'aspects. Des recherches théoriques et empiriques sur le sujet serait pertinentes.

Il serait très intéressant d'effectuer une nouvelle étude similaire à celle que nous avons menée mais en utilisant notre cadre applicatif Mock4Aj. Ainsi, il nous serait possible de déterminer si notre solution permet, réellement et dans la pratique, de simplifier l'écriture de tests unitaires d'aspects.

Concernant Mock4Aj lui-même, quelques primitives et cas ne sont pas actuellement couverts. Il s'agit généralement de cas triviaux qui pourraient être facilement ajoutés par de futurs utilisateurs étant donné que le projet est libre et ouvert. Une amélioration qui pourrait cependant requérir de la recherche et développement serait les tests pour la primitive « cflow » ainsi que l'instanciation des aspects.

L'annexe A présente des User Stories qui n'ont pas été prévues pour cette version mais qui pourraient constituer quelques pistes d'améliorations.

Annexe A

Liste des *User Stories* pour Mock4Aj

ID	Titre	Detail	Points
45	[M4AJ-01a] Choisir et isoler l'aspect à tester	Comme développeur, je dois pouvoir choisir quel(s) aspect(s) tisser dans quelle(s) classe(s) pour pouvoir injecter uniquement l'aspect sous test.	3
58	[M4AJ-01b] Créer un proxy tissé d'un mock existant	Comme testeur unitaire, je dois pouvoir tisser des mocks déjà créés. C'est utile car il est possible que j'aie déjà créé un mock avec un framework qui n'est pas supporté ou encore une sorte de mock spécial. Puisqu'il n'est pas possible de tisser une classe (le mock) déjà chargée par Java, il faut alors créer un proxy tissé qui va transférer (proxy) les appels vers le mock existant.	8
61	[M4AJ-01c] Utiliser AspectJ comme tisseur dynamique	Comme testeur unitaire, je veux pouvoir utiliser AspectJ dynamiquement afin de tisser mes mocks. Le tissage doit être dynamique, indépendant du LTW et ne requérir aucun Java Agent.	3
47	[M4AJ-03] Vérifier le comportement de l'aspect à l'exécution d'une méthode réelle	Comme testeur unitaire, je veux simuler l'exécution d'une méthode existante dans une classe pour vérifier si l'aspect testé se comportera correctement lors de l'exécution de cette méthode. Je peux simuler l'exécution d'une méthode qui doit être appareillée (l'aspect doit réagir) ou non (l'aspect ne doit rien faire).	1
48	[M4AJ-04a] Vérifier à l'exécution d'une méthode fictive (via interface)	Comme développeur, je veux simuler l'exécution d'une méthode qui n'existe pas pour vérifier la spécification du pointcut. Je peux le faire en créant une "fausse" interface et en utilisant un mock. Je peux simuler l'exécution d'une méthode qui doit être appareillée (l'aspect doit réagir) ou non (l'aspect ne doit rien faire).	1
66	[M4AJ-04b] Simuler l'exécution d'une méthode fictive (via contexte)	Comme développeur, je veux simuler l'exécution d'une méthode qui n'existe pas pour vérifier la spécification du pointcut. Je peux le faire en utilisant un contexte.	5
49	[M4AJ-05a] Simuler l'appel d'une méthode (call)	Comme développeur je veux simuler des appels de méthodes pour tester des pointcuts de type "call". Pour ce faire, je dois générer dynamiquement un Caller et mettre en place un mécanisme de sélection de la cible.	20
72	[M4AJ-05b] Écrire la facade statique pour le simulateur d'appels	Je dois écrire la façade statique pour le simulateur d'appesl dans Mock4Aj afin d'offrir une interface plus facilement utilisable. Voir [M4AJ-05a].	3
50	[M4AJ-06a] Simuler le contexte de l'appel d'une méthode (classe)	Comme développeur je dois pouvoir fournir un contexte d'appels d'une méthode pour pouvoir tester des pointcuts "call" où l'origine de l'appel est importante. Pour ce faire, je peux donner une classe d'origine ou un nom de classe. Pour l'instant, seul le "this" (type) et "within" (nom) sont supportés.	5
65	Documenter ce qu'il y a de fait dans le projet	Écrire le guide d'utilisation pour ce qui est fonctionnel et donner des exemples.	8
81	Modifications suite à la revue	Je dois faire quelques modifications après la revue de code	1
38	Projet: Monter l'infrastructure	Je dois établir l'infrastructure pour pouvoir débuter le projet.	3

34	Prototype: Créer des méthodes dans le mock	L'utilisateur doit pouvoir créer facilement des méthodes dans le mock afin de vérifier si le PC rapparie ou non.	3
35	Prototype: Donner le contexte de l'appel	L'utilisateur doit pouvoir spécifier le contexte de l'appel sur le Mock afin de recréer le contexte pour des PC avec des target, this, args, etc.	13
32	Prototype: Isoler les aspects	L'utilisateur doit pouvoir isoler les aspects pour pouvoir contrôler quel aspect sera tissé dans quel Mock.	8
36	Prototype: Supporter les packages	L'utilisateur doit pouvoir créer des Mocks dans d'autres packages pour tester les PC.	5
31	Prototype: Tisser dans les mocks en CTW	L'utilisateur doit pouvoir tisser des aspects dans des Mocks générés avec CGLib. Idéalement sans créer un Java Agent et il doit pouvoir contrôler quand et quel Mock doit être tissé. La solution doit être simple, expressive et facile à déboguer pour lui.	13
30	Prototype: Tisser dans les mocks en LTW	Je dois vérifier comment se comporte l'agent Java en LTW avec les Mocks créés par CGLib.	8
73	Refactoring du projet	Je dois réusiner le projet pour le rendre déployable.	3
71	Revue de code	Je dois demander à un collègue de faire une revue du code de Mock4Aj (Georges-Etienne Legendre).	2

Annexe B

Code source pour Mock4Aj

Le code est disponible sur Internet sous licence LGPL :

- Code source et tests : https://github.com/mock4aj/mock4aj
- Projet et documentation : https://code.google.com/p/mock4aj/

Bibliographie

[ABA04] Roger T Alexander, James M Bieman et Anneliese A Andrews : Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Colorado State University, Fort Collins, Colorado, 2004.

[ajt05] AJTE, 2005. (le site n'est plus disponible).

[Arm06] Deborah J Armstrong : The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, février 2006.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland et Dave Thomas : Manifesto for agile software development. http ://agilemanifesto.org/, 2001.

[BCP+05] J. Benn, C. Constantinides, H. K. Padda, K. H. Pedersen, F. Rioux et X. Ye : Reasoning on software quality improvement with aspect-oriented refactoring : A case study. *In* W. T. Tsai et M. H. Hamza, éditeurs : *IASTED International Conference on Software Engineering and Applications ;SEA*, pages 309–315, Phoenix, AZ, USA, novembre 2005. Acta Press.

[BH08] Marc Bartsch et Rachel Harrison : An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Journal*, 16(1):23–44, mars 2008.

[BL10] Felix-Antoine Bourbonnais et Luc Lamontagne : Using AOP for an academic agile project : A pilot study. *In 1st International Workshop on Empirical Evaluation of Software Composition Techniques (ESCOT 2010)*, Rennes, France, mars 2010.

[Bod04a] Ron Bodkin : Re : Announcing aUnit, a new unit testing tool for aspect, novembre 2004.

[Bod04b] Ron Bodkin : Re : Announcing aUnit, a new unit testing tool for aspect, novembre 2004.

[CG09] Lisa Crispin et Janet Gregory : *Agile Testing : A Practical Guide for Testers and Agile Teams.* Addison-Wesley Professional, 1 édition, janvier 2009.

[Col04] Adrian Colyer : Announcing aUnit, a new unit testing tool for aspects, novembre 2004.

[CP02] Alexander Chaffee et William Pietri : Unit testing with mock objects : Improve your unit tests by replacing your collaborators with mock objects. http ://www.ibm.com/developerworks/library/j-mocktest/index.html, novembre 2002.

[Dem79] Tom Demarco : *Structured Analysis and System Specification*. Prentice Hall PTR, facsimile édition, mai 1979.

[DPC01] J. Andrés Díaz Pace et Marcelo R. Campo : Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, octobre 2001.

[Fou] The Apache Software Foundation : Maven. http ://maven.apache.org/.

[Fow04a] Martin Fowler : Is design dead ?, mai 2004.

[Fow04b] Martin Fowler : Is design dead ?, mai 2004.

[Fow07] Martin Fowler : Mocks aren't stubs, janvier 2007.

[FP09a] Steve Freeman et Nat Pryce : *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1 édition, octobre 2009.

[FP09b] Steve Freeman et Nat Pryce : *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1 édition, octobre 2009.

[fSI01] International Organization for Standardization (ISO) : ISO/IEC 9126-1 :2001 software engineering – product quality – part 1 : Quality model. International standard, ISO, 2001.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson et John M. Vlissides : *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition édition, novembre 1994.

[HJ09] Uwe D.C. Hohenstein et Michael C. Jäger : Using aspect-orientation in industrial projects : appreciated or damned ? *In Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 213–222, Charlottesville, Virginia, USA, 2009. ACM.

[HK02] Jan Hannemann et Gregor Kiczales : Design pattern implementation in java and aspectJ. New York, NY, USA, 2002.

[jmo] jMock - an expressive mock object library for java. http ://www.jmock.org/.

[jsr04] JSR-000176 Java(TM) 2 SDK, standard edition 5.0 final release - JSR# 176, 2004.

[jun] JUnit. http ://junit.org/.

[KGRX08] Raffi Khatchadourian, Phil Greenwood, Awais Rashid et Guoqing Xu : Pointcut rejuvenation : Recovering pointcut expressions in evolving aspect-oriented software. Rapport technique COMP-001-2008, Revised March 2009, May 2009, août 2008.

[KM99] Mik Kersten et Gail C. Murphy : Atlas. *ACM SIGPLAN Notices*, 34(10):340–352, octobre 1999.

[KR02] Jonna Kalermo et Jenni Rissanen : *Agile software development in theory and practice*. Thèse de doctorat, University of Jyväskylä, Jyväskylä, 2002.

[KS04] C. Koppen et M. Störzer : PCDiff : attacking the fragile pointcut problem. *In European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[Les05] Nicholas Lesiecki : AOP@Work : unit test your aspects. http ://www-128.ibm.com/developerworks/java/library/j-aopwork11/, novembre 2005.

[Les11] Denis Lessard : Hydro-québec : 40% des contrats informatiques dépassent les budgets prévus. *La Presse*, 2011.

[Mar96a] Robert C. Martin : The dependency inversion principle, mai 1996.

[Mar96b] Robert C. Martin : The open closed principle, janvier 1996.

[Mar97] Robert C. Martin : Stability article. *Engineering Notebook, The C++ Report*, février 1997.

[Mar02a] Robert C. Martin : *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2nd édition, octobre 2002.

[Mar02b] Robert C. Martin : *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2nd édition, octobre 2002.

[Mar02c] Robert C. Martin : The single responsibility principle, février 2002.

[Mar08a] Robert C. Martin : *Clean Code : A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 1 édition, août 2008.

[Mar08b] Robert C. Martin : *Clean Code : A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 1 édition, août 2008.

[MBB07] Philippe Massicotte, Linda Badri et Mourad Badri : Towards a tool supporting integration testing of aspect-oriented programs. *Journal of Object Technology*, 6(1): 67–89, janvier 2007.

[MFC01] Tim Mackinnon, Steve Freeman et Philip Craig : Endo-testing : unit testing with mock objects. *In Extreme Programming Examined*, page 287–301. Pearson Education, 2001. ACM ID : 377534.

[moc] Mockito - simpler & better mocking. http ://mockito.org/.

[MS07] L. Madeyski et L. Szala : Impact of aspect-oriented programming on software development efficiency and design quality : an empirical study. *Software, IET*, 1(5): 180–187, 2007.

[Mul05] Joseph D. Muller : What are the benefits of aspect oriented programming to project iterations developed using agile processes ? *In Rensselaer at Hartford Computer Science Seminar 2005*, avril 2005.

[Nic04] Lesiecki Nicholas : Re : Announcing aUnit, a new unit testing tool for aspect, novembre 2004.

[Ora] Oracle : The reflection API. http ://download.oracle.com/javase/tutorial/reflect/index.html.

[Pag88] Meilir Page-Jones : *Practical Guide to Structured Systems Design.* Prentice Hall PTR, 2 édition, mai 1988.

[PCG+08] C. Pohl, A. Charfi, W. Gilani, S. Göbel, B. Grammel, H. Lochmann, A. Rummler et A. Spriestersbach : Adopting aspect-oriented software development in business application engineering. Brussels, Belgium, avril 2008.

[Sal12] Jean-Marc Salvet : Administration publique : 152 millions $ de projets informatiques «en difficulté». *La Presse*, 2012.

[SB05] Sérgio Soares et Paulo Borba : Implementing modular and reusable aspect-oriented concurrency control with AspectJ. Uberlândia, Brazil, 2005.

[Sco10] Scott W. Ambler : 2010 IT project success rates. *Dr. Dobb's*, 2010.

[Sin06] Maria Siniaalto : Test driven development : empirical body of evidence. Rapport technique, Information Technology for European Advancement, 2006.

[SLB02] Sergio Soares, Eduardo Laureano et Paulo Borba : Implementing distribution and persistence aspects with aspectJ. *ACM SIGPLAN Notices*, 37(11):174, novembre 2002.

[SM00a] Inc. Sun Microsystems : *Java Language Specification.* Second edition édition, 2000.

[SM00b] Inc. Sun Microsystems : Lexical structure. *In Java Language Specification.* Second edition édition, 2000.

[Soc04] IEEE Computer Society. : *Guide to the Software Engineering Body of Knowledge - SWEBOK.* IEEE Press, Piscataway, NJ, USA, 2004 version édition, 2004.

[Sta09] Standish Group : Chaos report 2009. Rapport technique, 2009.

[the] The aUnit tool website. http ://aunit.sourceforge.net/.

[Wam07] Dean WAMPLER : Aspect-Oriented design principles : Lessons from Object-Oriented design. *In Industry Track*, Vancouver, British Columbia, Canada, mars 2007.

[WHH08] Jan WLOKA, Robert HIRSCHFELD et Joachim HÄNSEL : Tool-supported refactoring of aspect-oriented programs. *In Proceedings of the 7th international conference on Aspect-oriented software development*, pages 132–143, Brussels, Belgium, 2008. ACM.

[XZ06] Tao XIE et Jianjun ZHAO : A framework and tool supports for generating test inputs of AspectJ programs. *In Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, Bonn, Germany, 2006. ACM.

[YSM+05] Yudai YAMAZAKI, Kouhei SAKURAI, Saeko MATSUURA, Hidehiko MASUHARA, Hiroaki HASHIURA et Seiichi KOMIYA : A unit testing framework for aspects without weaving. *In Workshop on Testing Aspect-Oriented Programs (WTAOP05)*, Chicago, USA, mars 2005.

[ZA07] Chuan ZHAO et Roger T. ALEXANDER : Testing aspect-oriented programs as object-oriented programs. *In Proceedings of the 3rd workshop on Testing aspect-oriented programs*, pages 23–27, Vancouver, British Columbia, Canada, 2007. ACM.

[Zha02] Jianjun ZHAO : Tool support for unit testing of aspect-oriented software. *IN PROC. OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development*, 2002.