République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri de Constantine
Faculté des Sciences de l'Ingénieur
Département d'Informatique

Ecole Doctorale de l'Est en Informatique
Pôle de CONSTANTINE

Option : **Génie Logiciel**

N° d'ordre :
Série :

Mémoire

Présenté en vue de l'obtention du diplôme de
Magister en Informatique

Thème :

Gestion de la qualité de service lors du développement des logiciels orientés aspect

Présenté par : ***BETTOU Farida***
Dirigé par : ***Prf. BOUFAIDA MAHMOUD***

Devant le jury

Président :
Dr. Zarour Nacereddine Maitre de conférence à l'université Mentouri de Constantine

Rapporteur :
Prf. Boufaida Mahmoud Professeur à l'université Mentouri de Constantine

Examinateurs :
Dr. Zeghib Nadia Maitre de conférence à l'université Mentouri de Constantine
Dr. Merniz Salah Maitre de conférence à l'université Mentouri de Constantine

Résumé

La qualité de service est un point de vue qui dépend du logiciel développé. Pour assurer une bonne qualité de service différentes techniques complémentaires ont été mises au point, nous nous intéressons principalement sur : Le design par contrat et Analyse de couverture, dont l'implémentation bénéficie particulièrement des avancées technologiques de la programmation orientée aspect.

Chaque technique assure la vérification d'un ensemble des exigences non fonctionnelles de qualité de service. Nous voulons bénéficier de tous les avantages des ces techniques. Pour cela nous proposons deux approches présentant l'implémentions du chaque technique à part dans un aspect, et la fusion des deux techniques dans un seul aspect.

Nous présentons une stratégie de transformation d'OCL contrats par AspectJ et une stratégie de transformation de méthode de test de couverture par AspectJ. Nous présentons aussi une deuxième approche d'intégration de design par contrat et test de couverture dans un seul aspect par AspectJ et nous donnons l'algorithme d'intégration.

Mots clés: qualité de service, design par contrat, Analyse de couverture, Programmation Orienté Aspect (POA), langage de contraintes objet (OCL), AspectJ.

Abstract:

The quality of service is a point of view depends on the software developed. To ensure good quality of service different complementary techniques have been developed, we focus mainly on: Design by contract and coverage analysis, whose implementation has particularly benefit of advanced technology of aspect-oriented programming.

Each technique provides verification of a set of non-functional requirements of service quality. We want to benefit of these techniques. For this we propose two approaches with the implementation of each technique in a separate aspect, and fusing the two techniques in one aspect.

We present a strategy for transforming OCL contracts with AspectJ and a strategy processing method of test coverage with AspectJ. We also present a second approach to integrating design by contract and test coverage in a single aspect in AspectJ and we give the algorithm of integration.

Keywords: QoS, design by contract, coverage analysis, Aspect Oriented Programming (AOP), Object Constraint Language (OCL), AspectJ.

الملخص:

نوعية الخدمة وجهة نظر تعتمد على البرمجيات المقدمة. لضمان نوعية جيدة من الخدمات هناك عدة تقنيات مختلفة و متكاملة تم تطويرها، ونحن نركز بشكل رئيسي على: التصميم بالعقد وتحليل التغطية، بما في ذلك تنفيذ فائدة خاصة من ناحية التكنولوجية المتقدمة، البرمجة جانبية المنحى.

كل تقنية تحقق من توفر مجموعة من المتطلبات غير وظيفية لجودة الخدمة. نريد أن نستفيد من كل مزايا هذه التقنيات. لهذا فإننا نقترح نهجين ،الأول نقترح فيه تنفيذ كل تقنية في جانب منفصلة ، والثاني دمج التقنيتين في جانب واحد.

نقدم استراتيجية لتحويل عقودOCL بلغة AspectJ واستراتيجية لتحويل طريقة اختبارالتغطية بلغة AspectJ.

نقدم أيضا في النهج الثاني إدماج التصميم بالعقد وتحليل التغطية في جانب واحد بلغة AspectJ ونعطي خوارزمية الادماج.

الكلمات الرئيسية : جودة الخدمة ، وتصميم من قبل العقد ، وتحليل التغطية ، البرمجة جانبية المنحى(POA) ،OCL ، AspectJ.

REMERCIEMENTS

Je présente mes sincères remerciements pour mon directeur de thèse, Monsieur le Professeur ***Boufaida Mahmoud*** *pour son aide et ses conseils très avisés tout au long de ma formation.*

Je lui témoigne ma profonde gratitude et ma reconnaissance. Son expérience et ses compétences ont facilité mon travail et m'ont profondément marqué.

Je tiens à remercier très sincèrement l'ensemble des membres du jury qui me font le grand honneur d'avoir accepté de juger mon travail.

Je remercie Monsieur ***Zarour Nacereddine*** *Maitre de conférence à l'université Mentouri de Constantine pour l'honneur qu'il me fait en acceptant la présidence de ce jury. Qu'il trouve donc ici l'assurance de ma profonde gratitude.*

Je tiens à exprimer toute ma reconnaissance à madame ***Zeghib Nadia*** *Maître de conférence à l'université de Constantine, ainsi que Monsieur* ***Merniz Salah*** *Maitre de conférence à l'université Mentouri de Constantine, pour avoir accepté d'être examinateurs de ce travail et pour le temps qu'ils ont investi à l'évaluer malgré leurs nombreuses obligations.*

Aussi, je remercie tout particulièrement mes parents pour leur encouragement et leur soutien.

A mes sœurs et mes frères

A toute ma famille

A tous mes amis (es)

Avec toute mon affection.

Sommaire

Chapitre III Une approche combinant le design par contrat et le test de couverture dans la programmation orientée aspect

Table des Figures

Introduction générale

Le but principal de l'ingénierie logicielle est de fournir une qualité du logiciel [27]. La qualité de service d'un logiciel permet de définir une information qualitative en plus de l'information quantitative sur le logiciel développé.

Le critère quantitatif est la conformité du logiciel développé par rapport au cahier des charges fourni par le client. Plusieurs techniques permettent de s'assurer de ce respect comme le test lors de son implémentation et conception par contrat. Ces techniques permettent de vérifier que les fonctionnalités spécifiées dans le cahier des charges sont bien intégrées dans le logiciel. Le critère qualitatif permet à l'utilisateur de définir des propriétés non fonctionnelles.

Ces propriétés servent à indiquer une qualité sur les fonctionnalités offertes par le logiciel [28]. On ne souhaite pas seulement savoir si le logiciel fait bien ce que l'on désire mais on veut de plus connaître des informations qualitatives sur ces fonctionnalités. Ces informations peuvent concerner la précision des résultats ou le temps d'exécution pris par une fonctionnalité. La qualité de service est un point de vue qui dépend du logiciel développé.

La qualité du Service (QoS) est devenue une majeure inquiétude publiée dans la conception, les opérations de réseaux et les systèmes distribués. Cela a deux raisons principales. En premier, plusieurs applications exigent des garanties de qualité du Service (QoS) pour leur opération adéquate. En seconde, toutes les fois que les utilisateurs sont chargés pour quelque service sa qualité réelle sera une inquiétude. [06]

Pour assurer une bonne qualité de service différentes techniques complémentaires ont été mises au point, [17] nous nous intéressons principalement sur : *Le design par contrat, Analyse*

de couverture, dont l'implémentation bénéficie particulièrement des avancées technologiques de la programmation orientée aspect.

L'implémentation de ces techniques dans la programmation orienté objet ou dans les autres paradigmes de programmation pose deux problèmes ; les fonctionnalités transversales et la dispersion du code. Ce phénomène de dispersion du code est un frein à la maintenance et à l'évolution des logiciels. Toute modification dont la manière d'utiliser un service entraîne des modifications nombreuses, coûteuses et sujettes à des erreurs.

Chaque technique assure la vérification d'un ensemble des exigences non fonctionnelles de qualité de service. Nous voulons bénéficier de tous les avantages des ces techniques. Pour cela nous proposons deux approches présentant l'implémentions du chaque technique à part dans un aspect, et la fusion des deux techniques dans un seul aspect en s'appuyant sur une meilleure séparation des préoccupations.

Dans ce mémoire, nous adopterons une organisation comportant quatre différents chapitres. Les deux premiers présentent l'état de l'art sur les techniques de gestion de qualité de service des logiciels et les concepts de la programmation orientée aspect.

- ✓ Dans le premier chapitre, nous analysons certains facteurs de qualité de service du logiciel, ainsi un aperçu des techniques de gestion de qualité de service des logiciels. Nous donnons les avantages et les inconvénients de chacune de ces techniques et une comparaison entre ces techniques de gestion de qualité de service des logiciels. Nous s montrons à travers quelques travaux que toutes les techniques de gestion de qualité de service des logiciels peuvent être bénéficiées des avantages de la programmation orientée aspect.

- ✓ Dans le deuxième chapitre, il sera consacre pour présenter une étude sur la programmation orientée aspect. Nous commençons cette étude par la présentation des problèmes résolus par la programmation orientée aspect. Puis nous introduisons les concepts de la programmation orientée aspect, en présentant aussi le langage AspectJ ainsi que ses concepts. Nous introduirons alors quelques exemples choisis de programmation par aspects écrits principalement avec le langage AspectJ vu comme une extension du langage Java. Ces exemples seront l'occasion de caractériser le modèle d'aspects sous jacent en précisant les notions de points de jonction, de coupes,

d’introductions. Nous terminons ce chapitre par la présentation de langage de modalisation des aspects au niveau conceptuel.

- ✓ Dans le troisième chapitre nous présentons l’approche globale de notre travail. Puis nous présentons la stratégie de transformation d’OCL contrats par AspectJ, cette transformation peut apparaître comme une génération de code, la stratégie de transformation de méthode de test de couverture par AspectJ. Et nous donnons les avantages et les inconvénients de cette première approche. Nous terminons ce chapitre par la présentation de notre deuxième approche d’intégration de design par contrat et test de couverture dans un seul aspect par AspectJ et nous donnons l’algorithme d’intégration et les avantages de cette intégration.

- ✓ Dans le quatrième chapitre nous allons le consacrer pour la présentation de nos approches de développement qui va être orientée aspect et donc implémenter une application de gestion bancaire par l’utilisation des aspects de gestion de qualité de service de nos approches.

Enfin nous concluons sur nos travaux avant de dégager des perspectives d’évolutions inspirées d’autres technologies.

A la fin de ce mémoire nous trouverons des annexes, détaillant les outils utilisé dans le quatrième chapitre.

Chapitre I

Etude sur les techniques de gestion de la qualité de service des logiciels

1. Introduction

La qualité de Service des logiciels (QoS) devient une question aussi importante que les systèmes sont plus ouverts, donc moins prévisible. Les informations de Qualité de Service (QoS) permettent aux concepteurs logiciels de spécifier quelles sont les garanties que doit fournir une application à l'exécution.

Le génie logiciel vise la production de logiciels de qualité. Ce chapitre présente un ensemble de techniques qui promettent d'améliorer sensiblement la qualité des produits logiciels.

Avant d'étudier ces techniques, nous devons clarifier leurs raisons d'être. La qualité du logiciel résulte de la combinaison de plusieurs facteurs. Ce chapitre analyse certains d'entre eux, ainsi un aperçu des techniques de gestion de qualité de service des logiciels. Nous donnons également les avantages et les inconvénients de chacune de ces techniques et une comparaison entre ces techniques de gestion de qualité de service des logiciels. Nous montrons à travers des travaux que toutes les techniques de gestion de qualité de service des logiciels peuvent être bénéficiées des avantages de la programmation orientée aspect.

2. La qualité de service des logiciels

Les systèmes sont élaborés à partir de besoins fonctionnels, c'est-à-dire en fonction de ce que l'on a besoin qu'ils fassent. Prenons l'exemple d'une machine à café, sa spécification fonctionnelle est de faire du café. Cependant, si la machine à café met trop de temps à faire le café, le client ne sera pas satisfait et la prochaine fois, il utilisera une autre machine qui prendra moins de temps. Cette spécification de temps sur le comportement fonctionnel de la machine est une spécification non fonctionnelle ou encore appelée qualité de service. [29]

La qualité d'un logiciel ne fait malheureusement pas souvent partie des processus du développement logiciel au sein des entreprises. Les entreprises qui ont peu de moyens ou pour lesquelles le temps de mise sur le marché est crucial négligent souvent cette étape du développement logiciel.

La qualité d'un logiciel est directement liée aux caractéristiques qui peuvent être mesurées à l'aide de tests et d'inspections. Plusieurs problèmes qui peuvent avoir des conséquences majeures sur le projet, peuvent être mis en évidence:

- Identification des exigences des usagers
- Implémentations des exigences des usagers
- Facilité d'utilisation et mise à jour de la documentation
- Maintenance du système

Ces activités, si elles sont négligées, peuvent causer l'échec d'un projet ou tout au moins l'affecter grandement. Le client peut se retrouver avec un système qui ne lui convient pas et ne l'utilisera pas [37].

Pour assurer une bonne qualité de services déférents techniques complémentaires ont été mises au point, [17] : *Le design par contrat*, *Analyse de couverture, Administration et supervision*

3. Facteurs de qualité de service des logiciels

Avant d'étudier ces techniques, nous devons clarifier leurs raisons d'être. La qualité du logiciel résulte de la combinaison de plusieurs facteurs.

La *qualité* d'un logiciel est définie par deux types de facteurs [36]:

- Des facteurs externes : perceptibles par les utilisateurs du produit (efficacité, fiabilité, etc)
- Des facteurs internes: perceptibles par les informaticiens (modularité, lisibilité, etc).

D'un côté, nous envisageons des qualités comme la vitesse ou la facilité d'utilisation, dont la présence, ou non, dans un produit logiciel peut être détectée par ses utilisateurs. Ces propriétés peuvent être appelées des facteurs **externes** de qualité.

Les autres qualités d'un produit logiciel, comme celles d'être modulaire ou facile à lire, sont des facteurs **internes**, seulement perceptibles aux informaticiens professionnels qui ont accès au texte source du logiciel. En dernier ressort seuls comptent les facteurs externes, mais ils ne peuvent être pris en compte que grâce aux facteurs internes. La plupart de ces facteurs reposent sur une confrontation aux spécifications.

Nous présentons quelques *facteurs externes* :

- *correction* ou *validité* : c'est l'aptitude du logiciel à réaliser exactement les fonctionnalités définies dans sa spécification ;
- *robustesse* : c'est l'aptitude du logiciel à réagir de manière appropriée aux situations anormales (donc non couvertes par les spécifications), à terminer élégamment sans déclencher de catastrophes ;
- *extensibilité* : c'est l'aptitude à s'adapter facilement aux changements de spécification ;
- *réutilisabilité* : c'est l'aptitude des éléments logiciels à servir à la construction de différentes applications.
  prenons l'exemple de Crash d'Ariane 5 causé par la *réutilisation* d'un composant logiciel et matériel qui convenait bien pour Ariane 4, mais contenait une contrainte de domaine «cachée» (fonction valide seulement si une certaine donnée restait < à 32768, ce qui n'était pas le cas pour Ariane 5).

- *compatibilité* : facilité à combiner des éléments logiciels entre eux. D'où l'importance que leur interface soit bien spécifiée ;
- *vérifiabilité* : propriété que le logiciel a de faciliter sa certification (déboguage, test, méthodes formelles, etc) ;
- *efficacité :* est la capacité d'un système logiciel à utiliser le minimum de ressources matérielles, que ce soit le temps machine, l'espace occupé en mémoire externe et interne, ou la bande passante des moyens de communication.[38]
- *ergonomie :* cousine de la ponctualité, est la capacité d'un système à être terminé dans les limites de son budget, ou en deçà. [38]
- *intégrité :* est la capacité que présentent certains systèmes logiciels à protéger leurs divers composants (programmes, données) contre les accès et modifications non autorisés. [38]
- etc.

Remarque : nous signalons les facteurs les plus importants :

- correction et robustesse assurent la fiabilité du logiciel.
- extensibilité et réutilisabilité assurent la modularité du logiciel.

Dans ce qui suit nous étudierons chaque technique à part.

4. Aperçu des techniques de gestion de qualité de service des logiciels

Cette section présente les différentes techniques de gestion de qualité de service des logiciels : design par contrat, test d'application, et administration et supervision, ainsi leurs avantages et leurs inconvénients.

4.1 Technique 1 *: Le design par contrat*

Avant d'introduire la notion de contrat, nous allons tout d'abord définir les notions importantes dans les contrats.

– ***Un service*** est une opération qui va être appelée par un composant et exécutée par un autre.

– ***Un client*** est un composant souhaitant utiliser un service.

– ***Un fournisseur*** est le composant qui va exécuter ce service.

a) Définition

Un contrat spécifie les droits et les obligations entre un client et un fournisseur de service. Les contrats déterminent les relations entre les différents acteurs.

On va distinguer deux opérations importantes dans les contrats qui sont deux façons distinctes d'exprimer un comportement complexe en terme de comportements plus simples:

– le raffinement qui permet la spécialisation des obligations contractuelles et des invariants de contrats, et

– l'inclusion qui décompose un contrat en sous-contrats plus simples.

b) Les niveaux de contrats

Le contrat vise à expliciter et contrôler la collaboration et le comportement entre les objets. Les contrats peuvent être classés en quatre niveaux où chaque niveau contient les propriétés des contrats de niveau inferieur.

b1)- Le contrat syntaxique

Il comporte les opérations que le composant peut exécuter, les paramètres d'entrées et de sortie qu'il doit avoir et les exceptions qui pourraient survenir lors de l'opération. Il permet de vérifier la conformité entre les interfaces fournies et requises du composant et facilite la vérification statique des composants.

b2)- Le contrat comportemental

Il spécifie le comportement des opérations en utilisant les pré- et post-conditions, pour chaque service offert et les classes d'invariants sur une déclaration d'opération:

Pré-condition de la méthode

- la méthode doit être satisfaite à la demande du service par le client.
- Elle spécifie la/les condition(s) nécessaire(s) pour qu'un client soit autorisé à appeler cette méthode ;
- en cas de violation de pré-condition, rien n'est garanti concernant l'exécution de la méthode (suivant les mises en œuvre : lancement d'exception, ou non exécution, ou exécution avec état résultant incohérent, etc) ;
- Elle caractérise l'ensemble des *états* (valeurs des attributs de l'objet et valeur des paramètres de la méthode) que la méthode devrait savoir traiter correctement ;

- couramment exprimée en fonction des paramètres formels de la méthode et des attributs de l'objet courant. [36]

Post-condition de méthode

- exprime la/les propriété(s) qui doi(ven)t être vérifiée(s) lorsque la méthode retourne (termine), si sa pré-condition était satisfaite au moment de l'appel ;
- elle doit être satisfaite par le fournisseur après la réalisation du service.
- caractérise l'ensemble des états dans lequel la méthode laisse les objets de l'application (l'objet sur lequel est appelé la méthode, mais aussi possiblement d'autres) ;
- elle doit souvent faire référence à des valeurs d'attributs avant appel ;
- dans le cas d'une fonction elle doit faire référence au résultat retourné par cette fonction. [36]

Invariant de classe

- spécifie des propriétés vraies dans l'ensemble des instances de la classe ;
- n'est donc exprimé qu'en fonction des attributs de la classe et de constantes ;
- pas de notion de "valeur avant appel" ;
- doit être établi (= rendu vrai) par chaque constructeur public, et doit être respecté par chaque méthode publique (= doit être vrai avant et après appel), et doit être vrai avant chaque appel de destructeur (si la notion de destructeur existe dans le langage utilisé) ;
- on peut considérer que c'est à la fois une pré-condition et post-condition de chaque méthode publique de la classe ;
- on se réserve le droit d'en dispenser les méthodes "helper" privées. [36]

La violation d'un contrat signifie un *bug* pour l'application. Si une pré-condition est violée, va indiquer une erreur au niveau du client c'est-à-dire que le demandeur n'a pas observe les conditions imposées pour des appels corrects alors que la violation d'une post-condition implique une erreur du côté du fournisseur, le service demandé a échoué. Plus on a de pré-conditions, plus la difficulté pour le client sera grande et plus la tâche sera facile pour le fournisseur.
Nous prenons l'exemple d'une fonction calculant une racine carrée d'un nombre réel qui peut être vérifiée de la manière suivante en pseudo code.

- Fonction calculant la racine carrée de la valeur x
    - Précondition : $x \geq 0$
    - Postcondition : *résultat* ≥ 0 et *résultat*$^2 = x$

La précondition assure que le développeur utilise la fonction correctement, alors que la postcondition permet au développeur de faire confiance à la fonction. [39]

Ces conditions peuvent être décrites en OCL (Object Constraint Language.

– Une pré-condition sera de la forme *pré : prédicat*,
– une post-condition sera de la forme *post : prédicat*. [36]

b3)- Le contrat de synchronisation

Le contrat comportemental considère les services comme atomiques. Le contrat de synchronisation, quant à lui, spécifie le comportement global des objets en termes de synchronisation entre les appels de méthodes.
Le but de ce contrat est de décrire les dépendances entre les services d'un composant tels que la séquence ou le parallélisme. Ce contrat garantit, quel que soit le client qui demande le service, que ce service sera correctement exécuté.
Il s'exprime sous forme de contraintes à respecter en temps réel et peut offrir de nombreux services, par exemple :
– l'accès à un objet peut être subordonné à l'obtention préalable d'un contrat (sécurité),
– son exclusivité garantit l'exclusion mutuelle (verrou),
– la liaison d'un ensemble d'objets, qu'ils soient locaux ou distants (groupes de communication, transparence à la localisation),
– l'intégration de la notion de contraintes de temps (*deadline*).

b4)- Le contrat de qualité de service

Une fois toutes les propriétés comportementales spécifiées, on peut essayer de quantifier le comportement voulu.

c) La programmation par contrat

Le paradigme de contrat a été très étudié par la communauté de génie logiciel. En effet, face à des modules logiciels devenus trop complexes pour comprendre aisément leur fonctionnement global, les contrats sont utilisés comme un sous-ensemble de la spécification du

système permettant de fixer un certain nombre d'invariants que le système s'engage à respecter ou que l'environnement doit respecter. [28]
Historiquement, la programmation par contrat a été introduite par Bertrand Meyer dans son langage Eiffel datant de 1985 [39]

> Le principe de cette programmation est de préciser ce qui doit être vrai à un moment donné de l'exécution d'un programme. Il ne faut pas penser que ce paradigme oblige à réaliser des tests effectifs des règles pendant l'exécution : ces tests ne sont qu'une des façons de s'assurer que les règles sont respectées.

Mais, contrairement aux méthodes de preuves de programmes, on ne va pas chercher à montrer explicitement que la spécification est bien réalisée par le programme. Cette partie est laissée à la discrétion du client et du fournisseur.

Néanmoins on met souvent en place des mécanismes de tests des règles pendant l'exécution pour vérifier leur validité. On peut utiliser en plus des tests unitaires pour vérifier les assertions de manière élégante. Mais cela ne permet en aucun cas d'être sûr que les règles sont tout le temps valides. En effet il faudrait, la plupart du temps, réaliser une infinité d'exécutions différentes pour vérifier tous les cas possibles. [39]

d) Langage de contraintes orienté objet (OCL)

Pour spécifier complètement une application, les diagrammes UML seuls sont généralement insuffisants, ils nécessité de rajouter des contraintes mais la langue naturelle manque de précision, compréhension pouvant être ambigüe, c'est avec OCL qu'UML formalise l'expression des contraintes. Il s'agit donc d'un langage formel d'expression de contraintes bien adapté aux diagrammes d'UML, et en particulier au diagramme de classes [40].

L'analyse et la conception par contrat permet la définition d'un accord formel entre une classe et ses clients, chaque partie exprime des droits et des obligations. Les contrats écrits dans l'OCL est connue pour être une technique utile pour préciser les préconditions et les postconditions des opérations et les invariants de classe dans un contexte UML, faisant de la définition d'analyse orientée objet ou de conception d'éléments plus précis tout en les aidant dans le teste et le débogage [40].

Dans [40], les auteurs notent que la combinaison de l'UML, avec OCL est, à l'heure actuelle, probablement le meilleur moyen de développer la bonne qualité et le haut niveau de modélisation, comme il a des résultats précis, sans ambiguïté, et les modèles compatibles.

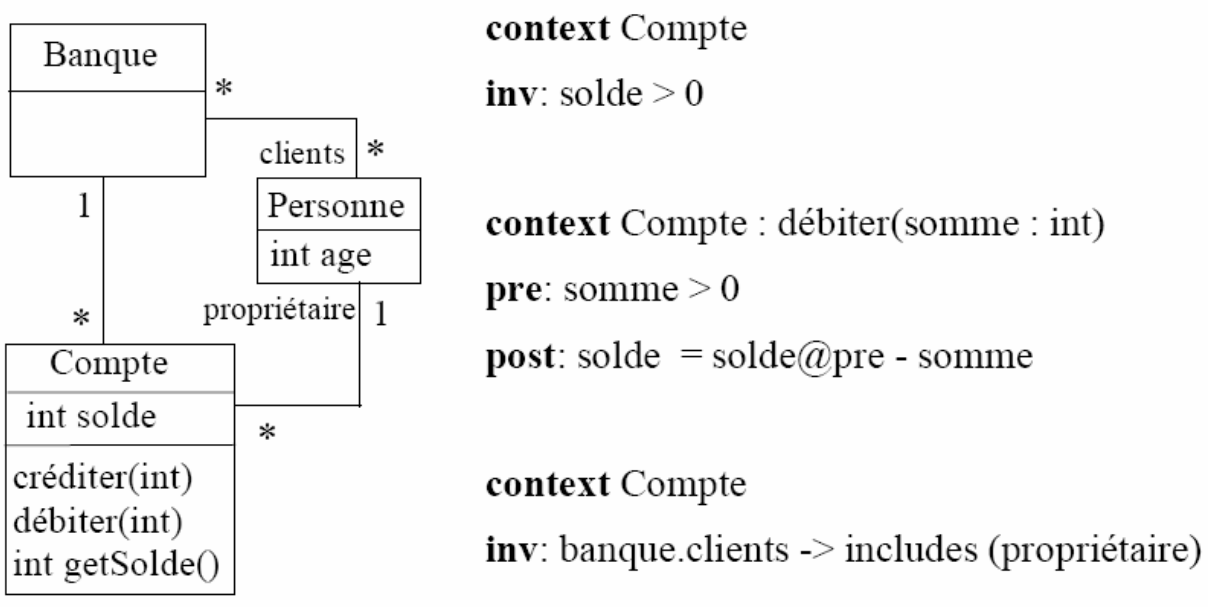


Figure I.1. Exemple d'usage d'OCL sur l'application bancaire

OCL existe depuis la version 1.1 d'UML et est une contribution d'IBM. OCL fait partie intégrante de la norme UML depuis la version 1.3 d'UML. Dans le cadre d'UML 2.0, les spécifications du langage OCL figurent dans un document indépendant de la norme d'UML, décrivant en détail la syntaxe formelle et la façon d'utiliser ce langage.

OCL représente, en fait, un juste milieu entre le langage naturel et un langage très technique (langage mathématique, informatique, …). Il permet ainsi de limiter les ambiguïtés, tout en restant accessible.

Notons qu'une expression OCL décrit une contrainte à respecter et ne décrit absolument pas l'implémentation d'une méthode.

Syntaxe de représentation de contrats par OCL dans UML

- **Contexte (*context*)**

Une contrainte est toujours associée à un élément de modèle. C'est cet élément qui constitue le contexte de la contrainte. Il existe deux manières pour spécifier le contexte d'une contrainte OCL :

En écrivant la contrainte entre accolades ({}) dans une note. En utilisant le mot-clef `context` dans un document accompagnant le diagramme (comme nous l'avons fait sur la Figure I.1).

context <élément>

`<élément>` peut être une classe, une opération, etc. Pour faire référence à un élément *op* (comme une opération) d'un classeur *C* (comme une classe), ou d'un paquetage, …, il faut utiliser les `::` comme séparateur (comme *C*`::`*op*).

- ***Invariants (inv)***

Un invariant exprime une contrainte prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence.

inv : <expression_logique>

`<expression_logique>` est une expression logique qui doit toujours être vraie.

- ***Préconditions et postconditions (pre, post)***

Une précondition (respectivement une postcondition) permet de spécifier une contrainte prédicative qui doit être vérifiée avant (respectivement après) l'appel d'une opération.

Dans l'expression de la contrainte de la postcondition, deux éléments particuliers sont utilisables :

1-Précondition :

pre : <expression_logique>

2-Postcondition :

post : <expression_logique>

`<expression_logique>` est une expression logique qui doit toujours être vraie.

`<nom_attribut> l`'attribut `résultat` qui désigne la valeur retournée par l'opération, et `<nom_attribut>@pre` qui désigne la valeur de l'attribut `<nom_attribut>` avant l'appel de l'opération.

e) Avantages *de design par contrat*

Les principaux avantages que peut offrir la technique de design par contrat sont :

- Une meilleure efficacité des tests du débogage, les contrats permettent de définir de manière beaucoup plus stricte les conditions d'exécution des différentes classes composant une application.
- Une meilleure documentation du code et réutilisation des composants, les contrats sont spécifiés clairement dans le code des composants donc facilement lisibles par sont exposées de manière explicite.
- Une meilleure gestion des erreurs en cas de non respect d'un contrant, plusieurs stratégies peuvent être mise en place afin de corriger ou contourner le problème.
- Moins de code à écrire c'est à dire :
  - une méthode n'a donc pas à vérifier sa pré-condition ;
  - après un appel de méthode un client n'a pas à vérifier la post-condition ;
- Il est reconnu que cette méthode permet quand même d'obtenir des logiciels de meilleure qualité et d'accélérer les phases de débogage.
- Réaliser un tel travail en amont permet non seulement de détecter plus rapidement les erreurs de conception, mais également, grâce aux invariants, celles liées à l'exécution. En cela, il s'agit d'un excellent complément aux tests unitaires [35].
- Les contrats peuvent être écrits dans un *langage de spécification* (dans le cas d'Eiffel, en Eiffel), en dehors du code fonctionnel ;
- code fonctionnel simplifié ;
- le compilateur génère le code de supervision des contrats correspondant aux différents appels de méthode ;
- possibilité de tout superviser, ou alors uniquement les pré-conditions, ou les pré- et post-conditions, etc : *monitoring*;
- génération automatique de documentation pour les contrats.
- Ce style de programmation oblige à réfléchir aux contrats avant de coder les méthodes et les classes, d'où idée plus claire de ce qu'elles font, et code de meilleure qualité.[36]

Avantages des contrats pour la qualité des logiciels

- Réduit le fossé entre le langage de spécification et langage de réalisation
- Garant de la qualité et facilite la réutilisation des composants [33]
- une meilleure séparation des préoccupations : si les approches par contrats cherchent avant tout à favoriser la réutilisation de modules logiciels, la déclaration explicite d'un certain nombre d'informations relatives à l'interaction permet de clarifier le code des méthodes. Ce code contient alors uniquement le code fonctionnel lié aux méthodes et n'est pas adapté par des parties de code vérifiant la correction des paramètres envoyés ou retournés au cours d'une interaction,[28]

f) Inconvénients *de design par contrat*

- La précondition d'un constructeur doit être vérifiée avant tout calcul dans le constructeur commence, ce qui signifie qu'il devrait se produire avant même que l'appel à la super constructeur.

  Les constructeurs ne sont pas hérités, et donc les préconditions d'un constructeur dans une superclasse ne s'appliquent pas aux constructeurs de ses sous-classes.
- Prévention de la récursivité infinie : les assertions peut êtres se référer à d'autres méthodes, y compris ceux de la même classe. Tout outil que les instruments du program de vérifier les assertions doit être attentif à ne pas entrer dans une récursivité infinie assertions au cours de la vérification.
- La compilation incrémentale : Parce que l'héritage des relations entre les classes affecte les assertions qui doivent être vérifiées, outils de conception par contrat ont besoin de savoir ces informations.
- Les anciennes valeurs dans les postconditions : ces valeurs doivent être calculé et conservé quelque part jusqu'à la fin de l'exécution de la méthode.
- Une grande partie de la partie technique de la programmation orientée aspect est sur l'instrumentation des programmes.

4.2 Technique 2 *: Test des logiciels*

Le test est une activité majeure pour l'obtention de logiciels de qualité. De bonnes pratiques, comme la vérification, les tests unitaires, les tests d'intégration tout au long du processus de développement, contribuent à la qualité du produit final. Le test occupe une part très importante du développement logiciel et constitue souvent l'activité principale de vérification.

Les tests dans le développement logiciel sont définis afin de vérifier une spécification ou un cahier des charges sur plusieurs cas d'exécution génériques. Le processus de test consiste à exécuter un programme dans le but de trouver les erreurs qu'il contient.

a) Définition

Un *test* est un ensemble de *cas à vérifiés* (état de l'objet à tester avant exécution du test, actions ou données en entrée, valeurs ou observations attendues, et état de l'objet après exécution), éventuellement accompagné d'une *procédure d'exécution* (séquence d'actions à exécuter).

La définition d'un test revient donc à définir cet ensemble. Différents types de test permettent de détecter différents types de défaut. Des *méthodes de spécification de test* ont été élaborées pour permettre une plus grande rigueur dans cette activité de définition. Les tests dans le développement logiciel sont définis afin de vérifier une spécification ou un cahier des charges sur plusieurs cas d'exécution génériques.

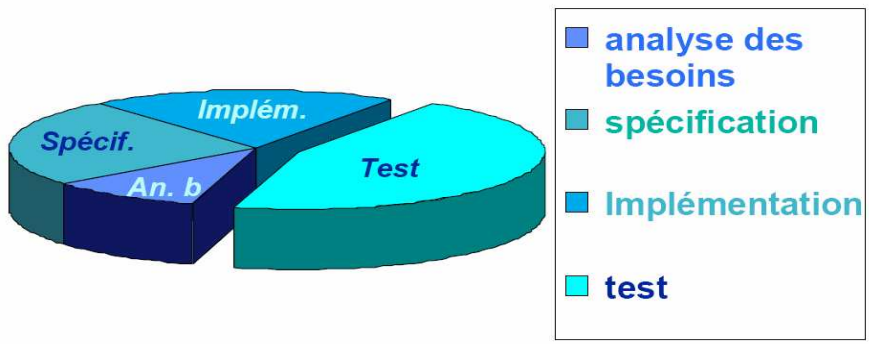


Figure I.2. Le test dans le cycle de vie des logiciels [45]

L'*activité de test* d'un logiciel utilise différents types et techniques de test pour vérifier que le logiciel est conforme à son cahier des charges (***vérification*** du produit) et aux attentes du client (***validation*** du produit). Elle est un des processus du développement de logiciel. Elle s'inscrit dans le contrôle de qualité, qui est indépendant du développement.

b) Catégories de tests (fonctionnels, structurels)

Toute application doit être testée afin de garantir à l'utilisateur final une qualité de service qui soit satisfaisable, à défaut d'êtres parfaite. Cette fonctionnalité constitue la brique de base

pour implémenter différents outils de test abordés ci-dessous. Il existe deux grandes catégories de test [17]: le test fonctionnel, et le test structurel.

➢ L'approche fonctionnelle ou boîte noire

On considère seulement la spécification de ce que doit faire le programme, sans considérer sa structure interne. On peut vérifier chaque fonctionnalité décrite dans la spécification. On s'appuie principalement sur les données et les résultats.

➢ L'approche structurelle ou boîte blanche

Basé sur l'analyse du programme [41]. Le test structurel vérifie l'application en s'attachant uniquement à la structure de son code source. L'objectif de ce test est d'identifier d'éventuels problèmes au sein même du code, tels que mauvaises pratiques de programmation, ou des variables et méthodes non utilisée. Un test structurel classique est le test de couverture, dans lequel l'exécution de l'application est analysée afin d'identifier les portions non couvertes par une campagne de test et donc potentiellement sources d'erreur [17].

Dans l'approche par *boîte blanche* on tient compte de la structure interne du module. On peut s'appuyer sur différents critères pour conduire le test comme :

c1) Le *critère de couverture des instructions* : le jeu d'essai doit assurer que toute instruction élémentaire est exécutée au moins une fois.

c2) Le *critère de couverture des arcs du graphe de contrôle*; le graphe de contrôle est un graphe qui résume les structures de contrôle d'un programme (cf. Figure I.3).

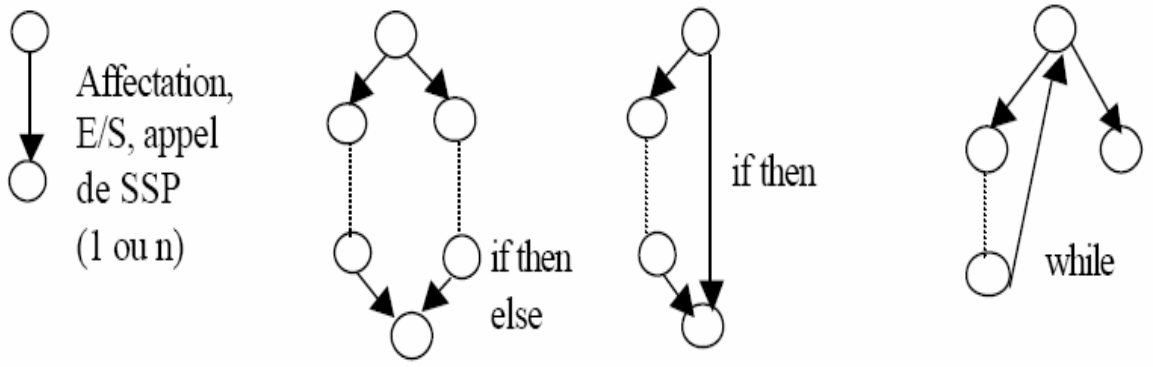


Figure I.3. Éléments d'un graphe de contrôle

La génération de cas de tests structurels complète la génération de cas de tests fonctionnels. Elle améliore la couverture de code et l'exécution de tels tests permet la définition de métriques sur le code.

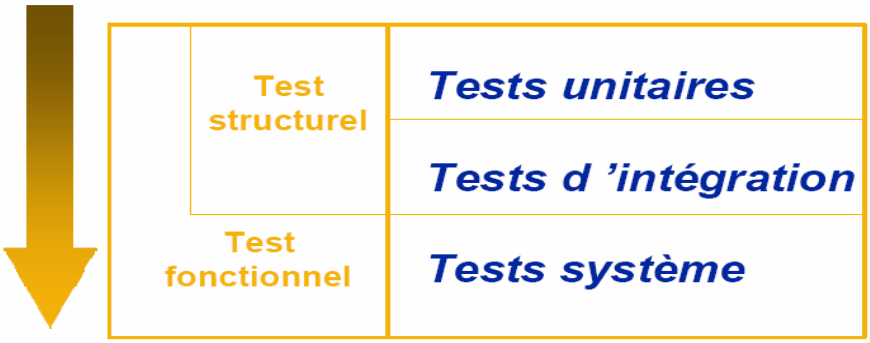


Figure I.4. Les catégories de tests

c) Test de couverture

Le test de couverture du code commence avec l'introduction d'instructions spéciales dans le code du programme, quelque fois par un préprocesseur, quelquefois par un modificateur de code objet, quelquefois en utilisant un mode spécial du compilateur ou de l'éditeur de liens, pour suivre tous les chemins possibles dans un bloc de code source et d'enregistrer, pendant leur exécution, tous ceux qui ont été parcourus. La génération des tests sur code source est également adaptée à la notion de couverture.

Les besoins en termes de qualité de code amènent de plus en plus de projets à s'intéresser et à coder des tests unitaires. Des indicateurs dits "de couverture de tests" sont utilisés en complément lors de la production de logiciels complexe.

Ces indicateurs représentent le pourcentage de code métier exécuté par les tests. Ils présentent plusieurs intérêts:

- De fournir une visualisation (sous forme de rapport html ou autre) du code exécuté et surtout du code non exécuté.

- De mieux maitriser le risque. Le code non testé est connu.

Plus cette couverture est importante pour des tests validant la spécification, plus fiable est la vérification du programme ou du logiciel.

Il y a deux critères de couverture :

- critère structurel : couvrir une portion précise du logiciel.

- critère fonctionnel : couvrir les fonctions du logiciel.

Le test de couverture comme expliqué précédemment, aussi appel analyse de la couverture du code, regroupe différentes méthodes permettant de vérifier que chaque partie d'une application est couverte par une campagne de test. On entend par partie couverte le fait que cette partie a été soit exécutée (instruction), soit accédée (variable, attribut, paramètre). Si une partie de l'application n'est pas couverte, c'est qu'elle est soit inutile, soit inaccessible, soit non prévue dans les scénarios d'utilisation. [17]

d) Avantages de Test des logiciels

- Le test est une activité de vérification et de validation, complémentaire à l'analyse statique (examen de documents)
- plus une faute est détectée tôt, plus on gagne beaucoup.
- il est reconnu que cette méthode permet quand même d'obtenir des logiciels de meilleure qualité et d'accélérer les phases de débogage.
- Les tests montrant la présence d'erreurs de programmation et la performance satisfaisante.
- Le test dans le cycle de développement d'un produit logiciel permet d'assurer le respect des exigences.

e) Inconvénients de Test des logiciels

- Il est en général impossible de tester tous les chemins possibles dans le code, il peut y en avoir des centaines, même dans une petite fonction de quelques dizaines de lignes. D'un autre côté, il n'est pas suffisant de s'assurer uniquement que les tests unitaires sont capables (éventuellement en plusieurs fois) de mettre à l'épreuve toutes les lignes de code. Si l'exécution d'un cas de test (pour un chemin allant du début à la fin du programme) prend une milliseconde, alors il faudrait plusieurs milliers d'années pour tester tous les chemins possibles de ce programme.
- Il n'existe pas d'algorithme apte à démontrer la correction d'un programme
- Les tests ont pour but de mettre en évidence les erreurs. Les tests peuvent *prouver la présence d'erreurs mais ne peuvent pas prouver leur absence.*

– Actuellement, le test dynamique est la méthode la plus diffusée et représente jusqu'à 60 % de l'effort complet de développement d'un produit logiciel.

4.3 Technique 3 *: Administration et supervision*

Bien qu'essentiels aux architectures n-tiers, l'administration et la supervision des applications sont encore des domaines mal pris en compte par les développeurs. [17]

Du fait de la centralisation et la mutualisation des traitements serveur, la panne d'un composant peut avoir des répercussions importantes, voire catastrophiques sur la disponibilité de l'application. Dans cette section nous présentons la technique d'administration et de supervision, une définition de JMX qu'est une extension qui consiste à permettre l'administration et la supervision des serveurs.

a) Définition

La supervision est une tâche de recueillir des mesures qui reflètent l'état d'un système [12]. Nous entendons la capacité à surveiller le fonctionnement de l'application [17]. être utilisés pour accroître la fiabilité des systèmes logiciels .L'administration est un ensemble de tâches de contrôle et de manipulation des systèmes informatiques [12], nous entendons la capacité à consulter et modifier les paramètres de l'application. [17]

b) Java Management Extensions (JMX)

Nous présentons dans cette section un exemple de service de supervision et d'administration. JMX (Java Management eXtension) [24] est un service de supervision et d'administration d'applications Java qui s'inspire de SNMP qui est le protocole d'administration de nœuds IP. JMX permet de superviser et d'administrer des ressources matérielles accessibles depuis une JVM comme des ressources logicielles s'exécutant sur une JVM. L'instrumentation des ressources se fait au moyen de composants appelés MBean (Manageable Bean). La spécification JMX définit l'ensemble des interfaces et classes nécessaires à cette instrumentation.[24]
JMX repose sur les deux grandes fonctionnalités suivantes pour remplir son rôle :

- Ouverture de composant Java (via une API spécifique) à l'extérieur pour les rendre accessibles à des outils d'administration et supervision ;
- Notification pour la surveillance de ces composants en cas de nécessité.[17]

L'architecture JMX comporte 3 couches [24] (voir figure I.5) :

• La couche ‘Instrumentation’ qui comporte les MBeans. Son rôle consiste à ouvrir aux couches (agent et services distribués) les composants destinés à êtres administrés ou supervisés. On parle alors de ressources gérables.

• La couche ‘Agent’ qui introspecte l'état des MBeans et exécute les opérations sur ces derniers, cette couche exploite les composants rendus gérables par la couche instrumentation et les rend accessible, via des adaptateurs, à l’extérieur de l’application c’est-à-dire à la couche service distribués. Elle propose plusieurs services utilisable par les services distribues, tels que le chargement dynamique de classes, le monitoring des ressources gérables…..

• La couche ‘Service distribués’ qui permet à des consoles distantes de l'interfacer avec le serveur de MBean.

° Les connecteurs déportent l'interface du serveur

° Tandis que les adaptateurs offrent des interfaces d'autres types (HTML/HTTP, SNMP, ...).

Cette couche regroupe les composants extérieurs à notre application interagissant avec la couche agent via des adaptateurs. Elle constituée d’outils d’administration et de supervision.

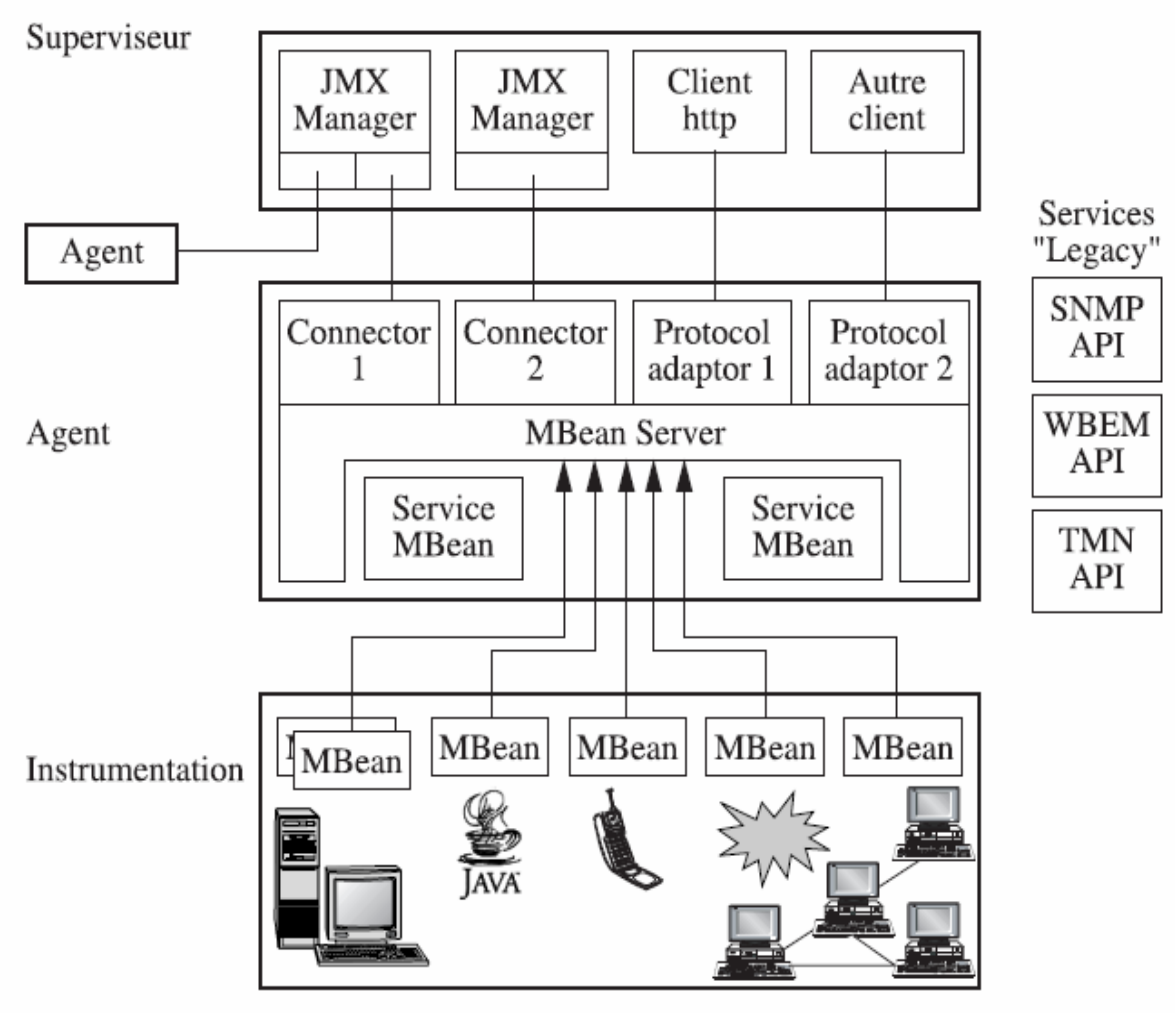


Figure I.5. L*'architecture générale de JMX [24]*

Le concept de base de l'architecture JMX est le MBean. Un MBean est littéralement un objet géré simple à implémenter qui représente une ressource pour ses besoins de supervision ou un service utile à la supervision. Concrètement, un MBean se traduit en un objet Java qui respecte un certain patron de programmation Java. Le MBean est utilisé pour instrumenter les ressources et forme la base du niveau instrumentation.

Les MBeans exposent une interface d'administration comportant un ensemble d'attributs consultables et/ou modifiables et un ensemble des opérations invocables. Il existe quatre types de MBean : standards MBeans, dynamique MBeans, ouverts MBeans et modèles Mbeans. Un standard MBean a une interface qui est définit statiquement. Le nom de l'interface implémentée doit être préfixé pour le nom de la classe d'implémentation et suffixé par "MBean".

```
public class NomClass implements NomClassMbean {
      ......}
```

Un dynamique MBean permet d'exposer dynamiquement ses attributs et ses opérations. La classe d'implémentation doit implémenter l'interface DynamicMBean. La liste des annotations est : @Description, @DescriptorFields, @DescriptorKey, @MXBean, @ManagedAttribute, @ManagedOperation, @MBean, @NotificationInfo, @NotificationInfos (package javax.management).

Dans l'exemple suivant, l'interface Mbean de la classe ObjetWeb définit deux types d'attributs en lecture seule(`MaxConnexionsn, IPHit`) ainsi qu'une méthode(`resetServer`) :

```
public interface ObjetWebMBean
{
public Integer getMaxConnexions() ;
public IPHitItem[] getIPHit();
public void resetServer(DateTime t, Integer m);
}
```

c) Avantages d'*Administration et supervision*

- Au travers de l'architecture et des interfaces JMX, la technologie Java démontre parfaitement son adéquation aux besoins des architectures de supervision.
- Elle offre une architecture d'agent souple et extensible fondée sur la technologie Java.
- La transparence et l'interopérabilité entre la couche supérieure: management application et la couche de base Management Système.
- Les ressources gérées et les fonctionnalités de management peuvent être ajoutées ou supprimées à tout moment.
- JMX a réussi à les simplifier au maximum sans cependant les restreindre trop fortement et a su mieux les mettre en œuvre d'un point de vue technologique, ceci de façon relativement simple.
- Les quatre types de *MBeans* et les mécanismes intégrés dans le serveur d'objets offrent une approche extrêmement facile à mettre en œuvre pour instrumenter des applications Java mais également grâce aux interfaces de programmation de services de supervision transversales, tout autre composant distant.

d) Inconvénients d'*Administration et supervision*

- Toutefois, il est complexe et lourde tâche de reconstruire pour les développeurs de systèmes logiciels existants en ajoutant des logiciels d'exécution de surveillance. [13]
- Pendant ce temps, les mécanismes de suivi d'exécution de logiciel sont principalement limités aux systèmes de surveillance centralisés de logiciels.[13]
- Pour l'administration des applications les paramètres technique et fonctionnel sont, dans le pire des cas codés en dur dans l'application, et il est impossible de les modifier sans recompilation de l'application .une solution classique à ce problème réside dans l'utilisation de fichier de propriétés lus par l'application à son démarrage. se pose alors le problème de la modification de ces paramètres à chaud, c'est-à-dire sans arrêter l'application. [17]

5 Comparaison entre les techniques de gestion de qualité de service des logiciels

Dans cette section nous présentons le résultat d'une étude comparative entre le design par contrat, test des logiciels et administration et supervision, nous avons défini des critères de comparaison.

Critères	***Le design par contrat***	***test de couverture***	***Administration et supervision***
Cycle de vie	Conception	Conception/ Implémentation	Implémentation
Concept de base	Précondition, Postcondition, Invariant	tests fonctionnels, structurels	Mbean
Langage de représentation	Langage de contraintes orienté objet (OCL)		Java Management eXtensions (JMX)
Préoccupation transversale	Oui	Oui	Oui
Langage de logiciel	Indépendant	Indépendant	orientée vers la supervision des applications Java

6. Techniques de gestion de qualité de service des logiciels et la POA

L'objectif de la programmation orientée aspect est basé sur la séparation des préoccupations. La contrainte principale est la dispersion des préoccupations transversales à travers le code métier [06]. Les techniques de gestion de qualité de service des logiciels peuvent être classées comme des préoccupations transversales.

Les principes de design par contrat sont peu présents dans les langages de programmation actuels mais peuvent êtres facilement implémentés avec les outils de la POA ainsi que les techniques des tests bénéficient de la capacité de la POA à instrumenter le code pour surveiller son exécution. [17]

L'administration et la supervision peuvent êtres vues comme des problématiques transversales à l'architecturée d'une application. Ces deux domaines couvrent un large spectre de composants sans pour autant participer activement à leur fonctionnalité première. Ce type de problématique correspond exactement à l'objectif de séparation des préoccupations de la POA. [17]

Pawlak et al [17] proposent un outil de test de couverture composé de deux éléments : l'enregistreur et le comparateur. L'enregistreur est chargé d'analyser l'exécution de l'application et d'enregistrer les méthodes appelées et les attributs accédés. Le comparateur confronte ces enregistrement avec le code source afin d'identifier les méthodes non appelées et les attributs non accèdes. L'enregistreur est implémenté grâce aux techniques de la programmation orientée aspect qui lui permettent d'intercepter les appels aux méthodes et les accès aux attributs. Mais pour la technique de teste de couverture, tous les appels aux les méthodes sont exécutés et l'erreur est connue seulement à la fin d'exécution. Aussi nous ne savons pas où est l'erreur. Cela montre qu'une erreur dans la conception par contrat est connue plus tôt que dans la technique de teste de couverture

Yishai et al [01] proposent Jose, un outil pour la conception par contrat en Java, qui utilise AspectJ alors que l'outil de Jose emploie, *after* advice ou *around.*

Ils décrivent dans [01], les décisions de conception de la mise en œuvre de José, et les caractéristiques d'AspectJ qui facilitent la mise en œuvre de contrat.

Rebelo [23] propose l'utilisation d'AspectJ à mettre en œuvre un nouveau compilateur Java Modeling Language (JML). Ce compilateur génère une instrumentation du bytecode, et la stratégie adaptée explore la mise en œuvre AspectJ JML's contrats (assertions).

Hamie [43] propose et discute des différentes stratégies de traduction de certains aspects des modèles de conception UML avec des contraintes OCL pour les classes et interfaces Java annotés avec JML affirmations qui sont la transformation des association, transformation des aggregation/composition transformation de généralisation.

Lionel et al [15] traitent la génération de traces d'exécution des systèmes distribués de Java, et étudie l'utilisation de la programmation orientée aspect (POA) pour l'instrumentation de code afin d'obtenir les informations nécessaires à l'exécution.

Dans [13], Jun et al présentent une nouvelle méthode, qui est distribuée dans les logiciels. C'est un mécanisme de surveillance qui est appliqué pour assurer la fiabilité des systèmes logiciels. Les auteurs utilisent la technique de la POA. Cette méthode permet de diminuer la pression du développement de développeurs.

Les travaux sur les différentes techniques de qualité de service du logiciel insistent sur la résolution des problèmes et les défauts de chaque technique à part. Ils fournissent des solutions et des approches qui utilisent la POA.

Notre approche décrit une stratégie pour la traduction de contrat OCL à un aspect, qui utilise AspectJ, nous basons sur ces deux stratégies [1] et [43] et nous intégrons la technique de design par contrat et de test de couverture, nous basons sur le travail de [01], [17] et [43].

7. Conclusion

Toute application doit êtres testée afin de garantir à l'utilisateur final une qualité de service qui soit satisfaisante. La POA nous permet d'instrumenter le code pour contrôler son exécution. Elle est particulièrement performante pour développer l'enregistreur puisqu'elle permet d'intercepter sans difficulté les appels aux méthodes afin de réaliser divers traitements. [17]

Mais nous trouvons que tous les travaux actuels n'intègrent pas ces techniques en un seul aspect. Notre approche consiste à combiner deux techniques le design par contrat et le test de couverture en utilisent la POA.

Nous avons présenté dans ce chapitre les facteurs de qualité de service des logiciels, ainsi un aperçu des techniques de gestion de qualité de service des logiciels et une comparaison entre ces techniques de gestion de qualité de service des logiciels . Nous avons montré à travers des travaux que tous les techniques de gestion de qualité de service des logiciels peuvent être bénéficiés des avantages de la programmation orientée aspect.

Chapitre II

Présentation de la programmation orientée aspect

1. Introduction

Du fait de ses nombreux atouts (encapsulation, polymorphisme, modularité, …), la programmation par objets constitue indéniablement une technologie importante pour aider à la construction d'applications complexes. En effet, elle favorise la *réutilisation* et permet ainsi la réduction des délais de développement et de maintenance.

Cependant, cette réutilisation n'est pas toujours aisée. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services donnés, mais nécessite également la prise en compte de différentes *propriétés non-fonctionnelles (ou d'infrastructure)* telles que la distribution et la persistance. Dans de telles applications, l'implémentation des services réalisés et celle des différentes *propriétés non-fonctionnelles* se trouvent intimement enchevêtrées. De ce fait, la réutilisation se trouve compromise. En effet, il n'est pas toujours possible de réutiliser les différents *aspects* (*i.e. services ou propriétés non-fonctionnelles)* d'une application, indépendamment les uns des autres [08].

Il est à noter que la programmation par aspects est indépendante de la programmation par objets. En effet, il est possible d'utiliser la programmation par aspects conjointement à d'autres paradigmes de programmation (fonctionnelle, impérative, …)[08].

Dans ce chapitre nous rappellerons d'abord les problèmes résolus pas la programmation orientée aspect. Puis nous introduisons les concepts de la programmation orientée aspect, nous présentons aussi le langage AspectJ ainsi que ses concepts. Nous introduirons alors quelques exemples choisis de programmation par aspects écrits principalement avec le langage AspectJ vu comme une extension du langage Java. Ces exemples seront l'occasion de caractériser le modèle d'aspects sous jacent en précisant les notions de points de jonction, de coupes, d'introductions.

Nous terminons ce chapitre par la présentation des langages de modalisation des aspects au niveau conceptuel.

2. Problèmes résolus par la programmation orientée aspect

Le but principal de l'ingénierie logiciel est d'obtenir la qualité logicielle. Les approches traditionnelles telles que les approches objet fournissent un support important de décomposition permettant d'offrir la réutilisation, et pour améliorer la qualité logicielle, grâce à l'encapsulation, le polymorphisme, l'héritage et la délégation. Cependant, quelques préoccupations, appelées préoccupations transverses dans la terminologie orientée aspect, entravent la réutilisation des modules. [27]

L'objectif de la POA est la séparation des préoccupations. La contrainte principale est la dispersion des préoccupations transversales à travers le code métier.[06]

Dans cette section nous expliquons les problèmes résolus par la programmation orientée aspect.

2.1. Fonctionnalités transversales

Une préoccupation est l'abstraction d'un but particulier ou une unité modulaire d'intérêt. Un système typique admet principalement deux types de préoccupations : des préoccupations fonctionnelles et des préoccupations techniques (non fonctionnelles). [30]

Dans [17], nous trouvons comme exemple un cas de contrainte d'intégrité référentielle : un objet client ne peut être supprimé s'il n'a pas honoré toutes ses commandes.

Comme la classe client n'est pas censée connaitre les contraintes imposées par les autres classes et que la classe commande n'a aucune raison de permettre la suppression d'un client, nous nous retrouvons face à un problème quant à la séparation indépendante des tâches. C'est ce que l'on appelle une fonctionnalité transversale (crosscutting concern).

Alors que le découpage des données en classes a pour but de rendre les classes indépendantes entre elles. Les fonctionnalités transversales telles que les contraintes d'intégrités référentielles viennent se superposer à ce découpage et brisent l'indépendance des classes. La programmation orientée objet ne fournit aucune solution, pour ce genre de problème.

2.2. Dispersion du code

En programmation orientée objet, on peut noter une différence entre les services offerts par une classe et les services utilisés (respectivement les méthodes et les appels de méthodes). Alors qu'il est facile de rassembler des services offerts dans une classe, il est impossible de rassembler l'utilisation d'un service.

Ainsi, l'implémentation d'une méthode est clairement localisée (puisqu'elle est dans une classe), alors que les appels vers celle-ci se retrouvent dispersés dans différentes classes (voir **Figure II.1**). Dés lors, la modification de la méthode est aisée mais pas la modification de sa signature. Une modification de la signature d'une méthode implique la modification de toutes les classes invoquant cette méthode, ce qui rend la maintenance et l'évolution du code difficile. [07]

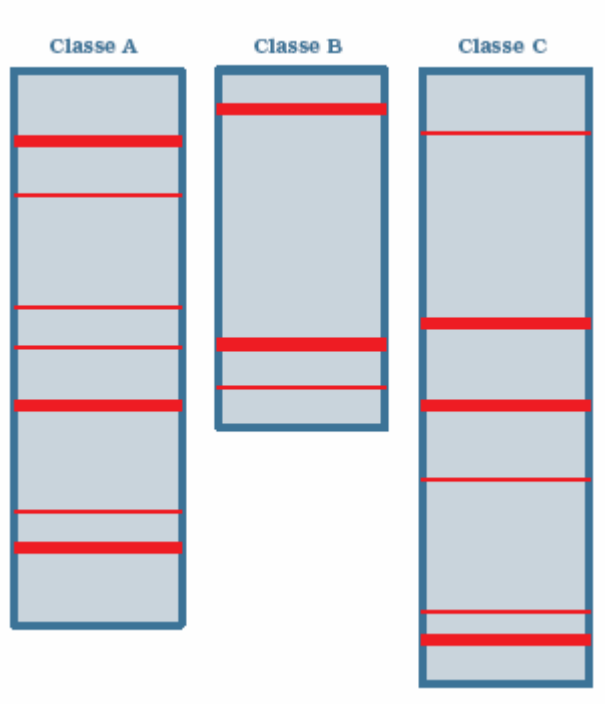


Figure II.1 – Dispersion du code d'une fonctionnalité

Outre le problème de mélange de code, la réutilisation d'une propriété non-fonctionnelle est d'autant plus difficile que sa définition se trouve dispersée. De ce fait, elle ne peut être isolée pour être réutilisée. En effet, une même propriété non-fonctionnelle peut toucher plusieurs objets. [08]

3. Découverte du monde des Aspects

Dans la présente section nous rappellerons les grandes lignes des paradigmes de programmation sur lesquels s'appuiera la programmation par aspects.

3.1. Historique

La programmation orientée aspect (POA) est un nouveau paradigme de programmation qui trouve ses racines en 1996, suite aux travaux de Gregor Kiczales et de son équipe au Centre de Recherche Xerox à Palo Alto[07] grâce à un papier publié dans les actes de la Conférence européenne sur la Programmation Orientée Objet en Juin 1997. L'équipe a également développé un langage populaire appelée AOP AspectJ, qui a acquis une grande popularité et l'acceptation de la communauté des développeurs Java [39]. La POA est donc une technologie relativement jeune.

3.2. Principe de POA

Une solution aux problèmes de mélange et de dispersion du code des propriétés non-fonctionnelles rencontrés avec la programmation par objets, consiste à séparer et découpler leurs définitions comme le veut le principe de la *separation of concerns*.

Partant de ce principe, l'équipe dirigée par Gregor Kiczales a formalisé cette séparation et les différentes étapes de réalisation des applications dans ce cadre pour aboutir au paradigme de programmation appelé *programmation par aspects (AOP : Aspects Oriented Programmation).* Ce paradigme de programmation consiste à structurer les applications en modules indépendants qui représentent les définitions de différents *aspects.*

Ces modules étant tous indépendants les uns des autres, il devient beaucoup plus simple de des coder, lors du développement, mais aussi pour les maintenir par la suite. Enfin et surtout, ces *aspects non-fonctionnels* pourront aussi être beaucoup plus facilement réutilisables.[08]

Il est important de souligner que la *programmation par aspects* utilise la technologie *objet* dans un cadre qui en conserve les bénéfices tout en palliant les limitations suscitées.

3.3. Quelques concepts de la programmation orientée aspect

Dans cette section, nous introduisons les concepts de ce nouveau paradigme. Nous expliquons quelques les constituants de l'aspect, comment les points de jonction, les coupes, le 'codes advice', 'Tissage d'aspects', 'Mécanisme d'introduction' .

a) Notion d'Aspect

La programmation orientée aspect propose l'utilisation d'aspects qui permettent d'implémenter des fonctionnalités transversales, ou dont l'utilisation s'avère dispersée dans le code, en intégrant aux classes des éléments (comme des méthodes ou des données) supplémentaires.

L'intégration de ces éléments supplémentaires dans les classes se fait au moyen de codes advices et de coupes qui permettent de spécifier le code de la fonctionnalité et où celui-ci va s'appliquer. Ainsi, les aspects résolvent le problème d'indépendance des classes puisqu'ils se chargent des fonctionnalités transversales et permettent de réduire la dispersion de code dont nous avons parlé grâce à une localisation de l'utilisation des services (voir **Figure II.2**)[07].

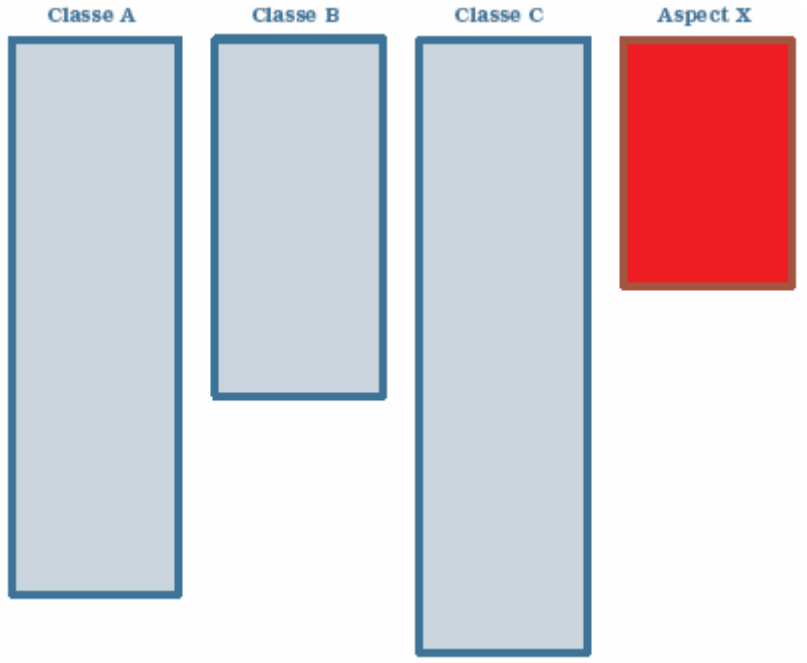


Figure II.2 – Localisation du code d'une fonctionnalité transversale dans un aspect

Cette nouvelle notion d'aspect, introduite par la POA, change la façon dans le développement d'une application à réaliser. Les fonctionnalités transversales sont expulsées du code métier.

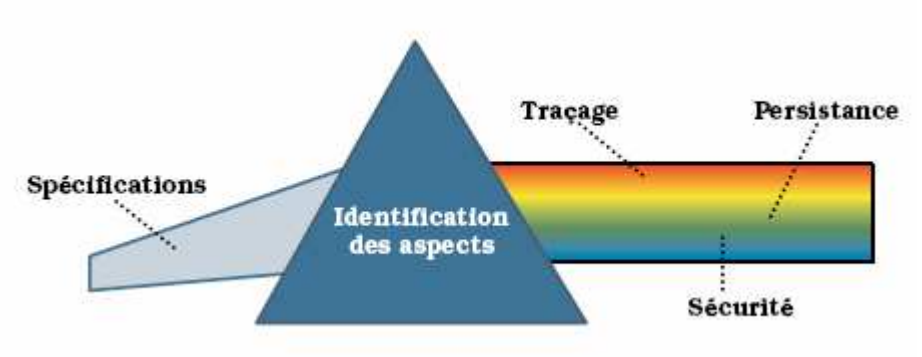


Figure II.3 – Identification des aspects d'une application

Les aspects identifiés peuvent être implémentés en parallèle. Ainsi, la partie sécurité d'une application pourra, par exemple, être confiée au service approprié de l'´equipe de développement, pendant que l'aspect persistance sera implémenté par un autre service. En prolongeant notre exemple, on peut imaginer que, deux mois plus tard, le service de sécurité trouve une meilleure technique de protection des données. Il lui suffira de modifier le fichier contenant l'aspect de sécurité pour modifier l'ensemble de programme. [07]

b) Notion de Point de jonction

Les aspects définissent les endroits du code où ils vont intégrer les fonctionnalités transversales. Cette spécification se réalise à l'aide de coupes qui sont elles-mêmes spécifiées à l'aide de points de jonction.

Un point de jonction est un point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés [17].Il représente un événement dans l'exécution du programme qui peut être intercepté pour exécuter un code advice correspondant. Pour cela, plusieurs types d'événements peuvent faire figures de points de jonction :

- **Méthodes :** Les méthodes sont les éléments principaux qui structurent l'exécution des programmes orientés objets. Les événements liés aux méthodes qui constituent des points de jonction sont l'appel d'une méthode et l'exécution de celle-ci.

- **Constructeur** : Les constructeurs peuvent être considérés comme des méthodes particulières. Il s'agit de méthodes invoquées lorsqu'un objet est instancié. Comme pour les méthodes, la POA permet d'intercepter cet événement.

- **Attributs :** La lecture et la modification d'attributs constituent également des points de jonction.

- **Exception** : Les événements de levée et de récupération d'exceptions peuvent aussi constituer des points de jonction. Le code à exécuter lors de la levée de cette exception sera défini dans un aspect.

c) Notion de Coupes

Une coupe désigne un ensemble de points de jonction [07]. Points de jonction et coupes sont conceptuellement fort différents : alors qu'un point de jonction représente un point dans l'exécution d'un programme, une coupe est un morceau de code défini dans un aspect. C'est à elle qu'est attribué le rôle de définir la structure transversale d'un aspect.

Dans le cas simples, une seule coupe suffit pour définir la structure transversale d'un aspect. Dans les cas plus complexes, un aspect est associé à plusieurs coupes.

En général, la définition d'une coupe passe par l'utilisation d'une syntaxe et de mots-clés. Chaque outil de la POA fournit sa propre syntaxe et ses propres mots-clés. Ces mots-clés identifiant des ensembles de points de jonction. Ces ensembles sont unis dans la coupe par des opérations ensemblistes comme par exemple : intersection et union.

d) Codes advice

Un aspect définit une fonctionnalité transversale. Comme nous l'avons vu, il spécifie le caractère transversal grâce aux coupes. La fonctionnalité est, quant à elle, spécifiée par des codes advices.[07]

Un code advice définit donc un bloc de code qui va venir se greffer sur les points de jonction définis par la coupe à laquelle il est lié. Un aspect nécessite souvent plusieurs codes advices pour caractériser la fonctionnalité transversale.

Il y a différentes manières de greffer un code advice sur un point de jonction. Le code advice spécifie lui-même la façon dont il souhaite s'intégrer aux points de jonction de sa coupe.Les trois principaux types de greffe de codes advices sont :

– Avant les points de jonction

– Après les points de jonction

– Autour des points de jonction

Pour une greffe d'un code advice autour des points de jonction de la coupe, il est nécessaire de spécifier la partie du code qui s'effectue avant le point de jonction et la partie qui s'effectue après. Les outils existants de la POA fournissent à cet effet un mot-clé qui permet d'exécuter le point de jonction : *proceed.*

e) Tissage d'aspects

Comme nous l'avons vu, les aspects définissent des morceaux de code et les endroits de l'application où ils vont s'appliquer. Un traitement automatique est donc nécessaire pour intégrer ces aspects dans l'application afin d'obtenir un programme fonctionnel fusionnant classes et aspects. Cette opération, représentée sur la **Figure II.4** , se nomme le tissage (weaving) et il est réalisée par le tisseur d'aspects (aspect weaver).[07]

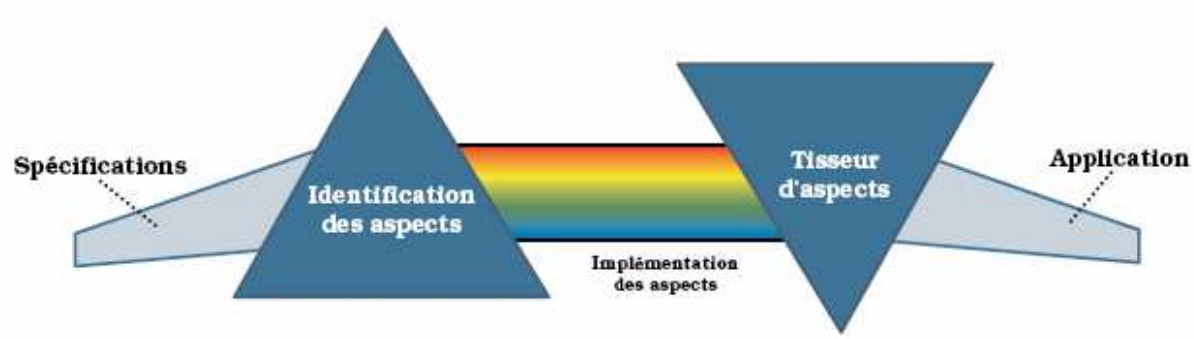


Figure II.4 – Tissage des aspects

On dit qu'un tisseur est statique lorsqu'il greffe réellement du code avant ou après la compilation. Avant la compilation, nous avons affaire à un tisseur qui modifie directement le code des classes cibles. Après la compilation, le tisseur doit faire son travail sur le bytecode.

Les tisseurs dynamiques quant à eux peuvent s'exécuter au chargement des classes en mémoire lors de la compilation Just-In-Time ou à l'exécution en interceptant les invocations de méthodes. [22]

Les avis des experts sont partagés sur ce sujet. Certains utilisent la robustesse et la performance des tisseurs statiques, d'autres la souplesse des tisseurs dynamiques. En réalité, tout dépend du besoin de l'application: si on veut tisser un aspect à la volée, en cours d'exécution de vos applications, seul un tisseur dynamique pourra vous satisfaire. Si les applications sont déjà un peu juste en termes de performances, ou si nous avons besoin de "tisser très fin" (avant ou après l'accès aux attributs, qu'il existe ou non des accesseurs pour ces derniers), alors il vous faudra s'orienter vers les tisseurs statiques [22]

f) Mécanisme d'introduction

Le mécanisme d'introduction de la POA permet d'étendre des classes en y ajoutant des éléments. Ces éléments sont essentiellement des attributs ou des méthodes, mais ce ne sont pas les seuls exemples. AspectJ propose notamment l'ajout d'interfaces Java et même l'ajout d'une superclasse.

A la différence de l'héritage en POO, l'introduction ne peut étendre les classes qu'en rajoutant de nouveaux éléments, il n'est donc pas possible de redéfinir une méthode, par exemple. Il est aussi important de remarquer que tous les éléments introduits ne peuvent être utilisés que par des aspects. En effet, l'application ne peut pas savoir à l'avance qu'un élément sera ajouté à une classe et donc en faire usage.

Néanmoins, si cela s'avère nécessaire, cette limite d'utilisation des éléments introduits peut être contournée en utilisant des fonctionnalités de réflexivité par exemple. Pour une introduction de méthodes, il suffit de demander au runtime toutes les méthodes d'une classe et celles introduites feront partie du lot récupéré. [07]

4. Aperçus sur le langage AspectJ

AspectJ est Le langage de POA le plus utilisé, il ajoute des mots clef au langage Java. Le langage Java est actuellement le plus utilisé pour la programmation par aspects ; et les deux projets les plus avancés sont **AspectJ** et **JAC**. Il s'agit en fait de tisseurs pour l'*aspect* avec un environnement de programmation. [08]

4.1 Aspects dans AspectJ

Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une application [07].

Des aspects pour AspectJ sont déclarés dans des fichiers **.aj**. Comme Java, les aspects peuvent être définis dans une classe. A la différence de Java, l'aspect ne doit pas avoir le même nom que la classe qui le contient ni celle que la classe Java auquel il sera appliqué.

Les aspects sont déclarés comme les classes:

```
public aspect monAspect {}
```
[21]

Un aspect est défini dans un fichier qui, porte le même nom que le nom de l'aspect. Ce fichier porte l'extension **.java**, mais il peut porter aussi l'extension **.aj** ou **.ajava** .

En général, **.aj** ou **.ajava** sont utilisés pour designer les fichiers aspects et **.java** pour designer les fichiers classes.

4.2 Points de jonction dans AspectJ

Ce sont des points particuliers dans l'exécution d'un programme. Un point de jonction peut par exemple définir un appel à une méthode d'une classe.

Les points de jonction, le concept essentiel en AspectJ, sont des points définis dans l'exécution d'un

programme. Cela peut inclure des appels à une méthode. Les points de jonction disposent d'un contexte qui leur est associé. Par exemple, un point de jonction d'appel à une méthode aura dans son contexte l'objet cible, ainsi que les éventuels arguments qui lui ont passés.

Bien que n'importe quel point d'exécution dans du programme puisse être défini comme un point de jonction, AspectJ limite les points de jonction à ceux qui peuvent être utilisés de manière systématique. [39]

a) Types de point de jonction

Les types de points de jonction fournis par AspectJ peuvent concerner des méthodes, des attributs, des exceptions, des constructeurs ou des blocs de code statique (**static).** Un dernier type concerne les exécutions de code advice. [32].

Les points de jonction, les plus utilisés, dans la POA sont :les méthodes, les attributs, les constructeurs et enfin les exceptions.

- **Méthodes :**

AspectJ définit deux types de points de jonction sur les méthodes : les appels de méthodes (‘*call’*) et les exécutions de méthode (‘*execution’*).

- Appel de méthode : *call(methexpr)* : Ce mot-clé identifie tous les appels vers des méthodes dont le profil correspond à la description donnée par ‘methexpr’.
- Exécution de méthode : *execution(methexpr)* : Ce mot-clé identifie toutes les exécutions de méthodes dont le profil correspond à l’expression ‘methexpr’.

La différence principale entre les types *call* et *execution* concerne le contexte dans lequel le se trouve l’application lors de l’exécution du code advice associé. Un point de jonction donnée par *call(methexpr)* correspond à l’appel de la méthode à methexpr. Il se trouve donc dans le code de la méthode appelante. Alors qu’un point de jonction donné par *execution(methexpr)* se trouve dans la méthode appelée. L’ordre d’exécution des différents codes advices serait le suivant :

1. Dans la méthode appelante : Exécution de la première partie du code advice associé au point de jonction *call*.
2. Dans la méthode appelée : Exécution de la première partie du code advice associé au point de jonction *execution*.
3. Dans la méthode appelée : Exécution de la méthode appelée.
4. Dans la méthode appelée : Exécution de la deuxième partie du code advice associé au point de jonction *execution*.
5. Dans la méthode appelante : Exécution de la deuxième partie du code advice associé au point de jonction *call*.

- **Attributs :**

Les points de jonctions de type *get* et *set* permettent d’intercepter respectivement les lectures et les écritures des attributs.

- Lecture d’attribut : *get(attrexpr)* : Identifie tous les points de jonction représentant la lecture d’un attribut dont le profil vérifie attrexpr.
- Modification d’attribut : *set(attrexpr)* : Ce mot-clé identifie toutes les modifications d’un attribut dont le profil est compatible à attrexpr.

Ces mots-clés sont intéressants pour des aspects qui souhaitent intercepter la modification de l’état d’un objet.

➢ **Constructeurs :**

AspectJ fournit deux types de points de jonctions, pour les constructeurs : *initialization* et *preinitialization*.

1. Exécution de constructeur : *initialization(constrexpr)* : Identifie toutes les exécutions d'un constructeur dont le profil vérifie constrexpr.

2. Exécution de constructeur hérité : *preinitialization(constrexpr)* : Ce mot-clé identifie toutes les exécutions d'un constructeur hérité dont le profil correspond à l'expression constrexpr.

L'exécution d'un constructeur hérité se fait en Java avec l'aide du mot-clé **super(..)** où **..** représente les paramètres fournis. Java impose que ce genre d'appel soit la première instruction d'un constructeur. Alors que *initialization* identifie l'exécution de constructeurs, *preinitialization* identifie l'exécution de constructeurs appelés via cette instruction. Il est intéressant de remarquer qu'il y a une possibilité d'intercepter les appels de constructeur avec le mot-clé *call.*

Le profil methexpr identifie les méthodes de nom *new*. AspectJ ne permet pas la définition d'un code advice de type *around* sur ces points de jonction.

➢ **Exceptions :**

Récupération d'exception : *handler(exceptexpr)* : Ce mot-clé identifie toutes les récupérations d'exception dont le profil vérifie exceptexpr. Il s'agit donc, en Java, de l'exécution des blocs *catch*. AspectJ ne permet pour le moment que l'utilisation de codes advices de type before sur les points de jonction de récupération d'exception.

AspectJ introduit d'autres de point de jonction, comme par exemple :

- Initialisation de classe : *staticinitialization(classexpr)* : Ce mot-clé identifie l'exécution des blocs statiques d'initialisation d'une classe correspondant à l'expression classexpr.

Exécution de code advice : *adviceexecution()* : Identifie toutes les exécutions d'un code advice. Il ne requiert pas d'expression en paramètre. Il identifie simplement toutes les exécutions de code advice. Il doit donc être utilisé avec précaution car il peut vite mener à des boucles infinies dans l'exécution du programme.

En effet, le code advice exécuté lors de l'exécution d'un code advice appartient, lui aussi, à l'ensemble défini par adviceexecution. Ce mot-clé doit donc être utilisé en parallèle avec des mots-clés de filtrage.

b) Définition des profils

Tous les mots-clés, présentés dans la sous-section des types de point de jonction nécessitent un paramètre, qui est une expression permettant, de spécifier un profil, et de filtrer l'ensemble de points de jonction donné par les mot-clé. L'expression précisant le profil des éléments peut faire usage de quantificateurs (appelés wildcards). Ces wildcards sont ***, .. et +**. [*32*]

4.3 Coupes dans AspectJ

Désigne un ensemble de points de jonction. [32]C'est un ensemble de points de jonctions corrélés (ensemble des endroits de mélange de code concernant la même fonctionnalité). Un pointCut est une référence à un point de jonction particulier ou un groupe de points de jonctions dans un programme où on souhaite agir. Un pointCut se compose de deux parties, séparées par un deux-points. On met dans la partie gauche le pointcut qui sera défini par un nom, et éventuellement une liste de paramètres.

Dans la partie droite indique les points de jonctions que le pointcut doit considérer. [21]
Exemple :

```
call(* aop.aspectj.MaClass.*(..)) && !call(*
      aop.aspectj.MaClass.*get*(..))
```

Représente l'ensemble des appels de méthode, dont le nom ne contient pas get, appartenant à la classe MaClass (dans le package aop.aspectj).

Les primitives suivantes permettent de décrire un pointcut. Elles peuvent êtres combinées par des opérations booléennes et leurs arguments sont des expressions régulières.
Par rapport à l'exécution du code en lui même :
– **call, execution** : permettent de décrire l'interception d'un appel de méthode, du point de vue, respectivement, de l'appelant et de l'appelé ;
– **initialization, preinitialization, staticinitialization** : interception des points-clés de l'initialisation d'une classe ;
– **get, set** : interception des accès aux attributs des instances, en lecture et en écriture ;
– **handler** : interception des levées d'exceptions « catchées ».
Par rapport aux valeurs des variables en présence :
– **this** : permet de définir des conditions sur l'instance courante sur laquelle est appelée la méthode en cours d'exécution ;

– **target** : idem que **this** mais dans le contexte de l'appelant, sur l'instance sur laquelle l'appel est effectue.
– **args** : permet de spécifier un pointcut dépendant des arguments d'appel de la méthode.
Par rapport à l 'exécution du code greffé: **adviceexecution, within, withincode, cflow, cflowbelow** et if permettent de spécifier des pointcuts dont la validité dépend du contexte d'exécution. Par exemple, spécifier un pointcut valide sauf dans le cas où il interviendrait dans le contexte d'exécution d'un code greffé. [06]
Exemples

```
call(void Point.setX(int));
```

Ce pointcut désigne l'ensemble des appels à la méthode setX de la classe Point prenant un unique paramètre de type int et retournant void.

```
methods(): call(* Point.*(..));
```

Ce pointcut désigne l'ensemble des appels à toutes les méthodes de la classe Point, quelque soit leur type de retour ou leurs arguments.

4.4 Codes Advices

Ce sont des portions de code qui sont exécutées lorsque certaines conditions sont rencontrées. Par exemple, un advice peut afficher un message avant d'exécuter un point de jonction. [39]
Les advices sont les blocs de code qui sont exécutes lors de l'interception d'un instant d'exécution du programme par un pointcut. Il existe trois modes d'exécution des advices :

– ***before*** : le compilateur insère un appel à l'advice avant le code correspondant au pointcut.
– ***after*** : le compilateur insère un appel à l'advice après le code correspondant au pointcut.
– ***around*** : le compilateur insère un appel à l'advice à la place du code correspondant au pointcut et fournit au programmeur un moyen d'appeler le code original avec les arguments qu'il souhaite et de récupérer l'éventuelle valeur de retour.

a) Type before

Before. Le code de l'advice est appelé avant l'exécution du code associé au pointcut. Ce mode n'offre pas de contrôle sur le comportement du code associé au pointcut. On peut néanmoins avoir accès aux valeurs des paramètres d'appels et du contexte de l'appelant si elles ont été exposées par la définition du pointcut.

Exemple :

```
/* définition de la coupe */
pointcut ex_coupe(): call(* *..*(..));
/* définition du code advice */
before(): ex_coupe() {
System.out.println("Avant un appel de méthode");}}
```

Ce code advice écrit le string "Avant un appel de méthode" avant chaque appel de méthode de l'application.

b) Type after

Le code de l'advice est appelé après l'exécution du code associé au pointcut. Ce mode offre les mêmes possibilités que les advices « before ». Il peut, en plus, être modulé selon la manière dont s'est terminé le code associé au pointcut :

– Le code s'est terminé normalement ;

– Le code s'est terminé anormalement, une exception a été levée. Il est possible de le moduler en fonction du type de l'exception ;

– Le code s'est terminé, quelle que soit la manière.

Exemple :

```
/* définition de la coupe */
pointcut ex_coupe(): call(* *..*(..));
/* définition du code advice */
after(): ex_coupe() {
System.out.println("Après un appel de méthode");
}
```

c) Type around

Le code de l'advice est appelé à la place du code associé au pointcut. Le programmeur dispose de la fonction *proceed* pour déclencher l'appel à ce code remplacé avec les paramètres qu'il souhaite. Le code de l'advice remplaçant le code associé au pointcut,
il doit se comporter comme lui, du point de vue de ce qui est retourné.

Ce mode d'appel est le plus flexible. Il permet de faire du filtrage sur les paramètres d'appel et sur la valeur de retour du code associé au pointcut. Il est tr`es confortable pour la réalisation de wrappers ou de vérificateurs de spécifications.

Exemple :

```
Object around(): ... {
          System.out.println("avant le point de jonction");
          Object ret=proceed();
          System.out.println("après le point de jonction");
          return ret;    }
```

d) Type after returning

Les codes advices de type *after returning* s'exécutent à chaque terminaison normale d'un des points de jonction de la coupe.

Exemple :

```
public aspect exemplep {

     after() returning (double d): call(double MaClass.add (..))
{
          System.out.println("add a retourné : " + d);
```

e) Type after throwing

Les codes advices de type *after throwing* s'exécutent à chaque terminaison anormale d'un des points de jonction de la coupe.

Exemple :

```
public aspect exemplep {

     after() throwing (Exception e): call(double MaClass.add
(..)) {
          System.out.println("add a levé l'exception : " + e);
```

4.5 Introspection de point de jonction

Nous avons vu qu'un code advice est exécuté dés qu'un point de jonction appartenant à sa coupe est rencontré pendant l'exécution. Le code advice ne sait donc pas à priori à quels endroits il sera exécuté et dans quelles conditions puisque sa coupe détermine un ensemble de points de jonction. A cet effet, certains outils de POA tels que AspectJ propose un mécanisme d'introspection de point de jonction.

Il s'agit donc d'obtenir des informations sur le code qui a provoqué l'exécution du code advice ou, autrement dit, inspecter le point de jonction pour en extraire ses caractéristiques.

On peut ainsi, par exemple, en extraire son type, sa signature, son emplacement dans le code ou encore, dans le cas d'un point de jonction de type méthode, les arguments fournis à celle-ci, l'objet cible ou l'objet source de l'appel.

4.6 Mécanisme d'introduction

Il s'agit d'étendre le comportement en ajoutant des éléments. Les catégories d'éléments peuvent êtres ajoutés concernent: l'attribut, la méthode, le constructeur, la classe héritée, l'interface implémentée et exception non conditionnelle à l'exécution d'un point de jonction [32].

a) Attribut

Comme nous l'avons dit, un aspect déclare des éléments pour le compte d'une classe. Cette déclaration se fait comme dans une classe si ce n'est qu'il faut préfixer le nom de l'attribut par le nom de la classe visée. Sans ce préfixe, l'attribut serait un champ de l'aspect. L'aspect suivant ajoute un attribut id à la classe Class :

Exemple :

```
public aspect AddId {
     private int Class.id;
     }
```

b) Méthode

L'introduction de méthode est semblable à l'introduction d'attribut. Elle se réalise comme la définition d'une méthode au sein d'une classe, sauf que l'on ajoute comme préfixe le nom de la classe cible.

Exemple :

```
public aspect AddId {
     private int Class.id;
     public int Class.getId() { return id; }
     }
```

c) Constructeur

Un constructeur est introduit exactement comme est introduite une méthode. Pour le mécanisme d'introduction d'AspectJ, un constructeur est juste une méthode dont le nom est new.

Exemple :

```
public aspect exemplep {

     public aspect AddId {
          private int Class.id;
          public int Class.getId() { return id; }
          public Class.new(int id) {
          this.id=id;
          }
          }
```

d) Interface implémentée

Le mécanisme d'introduction d'interface implémentée est similaire à l'introduction de classe héritée que nous venons de voir. Elle se réalise à l'aide du même mot-clé mais l'introduction d'interface implémentée est sans condition.

Il est toujours possible d'ajouter une interface à une classe. L'exemple suivant ajoute l'interface Interf à la classe Class et à toutes ses sous-classes, mais également `a toutes les classes dont le nom contient foo.

Nous donnons un exemple :

```
public aspect interface {
    declare parents: Class+ implements Interf;
    declare parents: *foo* implements Interf;
    }
```

4.7 Héritage dans la POA

AspectJ permet également de modifier la hiérarchie d'héritage des classes. A l'aide du mot-clé *declare parents*. AspectJ offre la possibilité de rendre une classe héritière d'une autre. Mais cette introduction n'est pas sans condition, il se peut en effet que la classe visée hérite déjà d'une surclasse. Dans ce cas, l'introduction ne sera possible que si la nouvelle surclasse est une sous-classe de l'ancienne surclasse.

Le mécanisme d'introduction d'interface implémentée est similaire à l'introduction de classe héritée. Elle se réalise à l'aide du même mot-clé mais l'introduction d'interface implémentée est sans condition. Le code suivant montre un exemple de cinq types d'éléments du mécanisme d'introduction.

```
public aspect MonAspect {

declare parents: MaClass1 extends MaClass2;
declare parents: MaClass3 implements MonInterface;

private int MaClass.id;
public int MaClass.add () {id=id+1 ; return id; }
public MaClass.new(int id) {this.id=id;}
}
```

5 Langage de modélisation des aspects

Le terme POA est en général lié à la phase d'implémentation, et puisque l'aspect s'intéresse à la séparation des préoccupations transversales, il sera intéressant d'identifier les aspects au niveau des phases préalables à la phase de programmation.

Plusieurs travaux ont été proposés pour définir l'aspect (comme entité) dans les phases d'analyse et de conception. Ils se sont intéressés à la séparation des préoccupations transversales (aspects) tout au long du cycle de développement de logiciel, et en particulier au niveau de la phase de conception.

La plupart des approches existantes proposent, pour la définition de leurs concepts et relations Aspect, d'étendre le metamodèle d'UML. Elles se distinguent cependant par leurs façon de faire : extension par stéréotypage (stereotyping), par metamodélisation (metamodeling),ou proposition d'un profile UML.[30]

En effet, tout comme les modèles et langages de programmation par aspects actuellement proposés sont très généralement construits comme des extensions, ou du moins des évolutions, de modèles et de langages de programmation par objets, la plupart des travaux portant sur les techniques de conception et de modélisation par aspects s'appuient très fortement sur les résultats acquis dans le cadre de l'approche Objet. En ce sens, UML le langage standard de modélisation par objets, est le plus souvent considéré comme point de départ a la définition d'un langage de modélisation par aspects, comme nous pouvons le constater d'ailleurs a travers l'analyse des principales propositions discutées ci-dessus. Ce choix est également justifie par plusieurs autres avantages d'UML [30] :

- UML, facilite l'expression et la communication d'une variété de modèles de conception qui permettent de modéliser de manière claire et précise la structure et le comportement d'un système d'information, indépendamment de toute méthode de développement ou de tout langage de programmation par objets .
- UML se base sur une syntaxe abstraite et une sémantique définissant un ensemble de concepts et leurs relations, qui forment les briques de base de la modélisation Objet. Il offre en plus une notation graphique dont la syntaxe concrète est à la fois simple, intuitive et expressive, pour pouvoir communiquer facilement les différents modèles de conception .
- UML est fondé sur un méta modèle qui décrit de manière formelle tous les éléments de modélisation d'UML, qui sont eux-mêmes modélisés avec UML. Cette définition

récursive, appelée méta modélisation, présente un double avantage: elle permet de classer les concepts et relations par niveaux d'abstraction, et aussi de faire la preuve de la puissance d'expression de la notation UML, capable entre autres de ce représenter elle-même .

- UML facilite l'interopérabilité: les modèles UML peuvent être facilement spécifiés sous des formes textuelles conformes à des standards OMG telles que XML et XMI, etc., afin de faciliter l'exploitation de leurs données.
- UML se veut un langage de modélisation non fermé, suffisamment général, extensible, et adaptable à plusieurs domaines d'applications spécifiques. Il se base en général sur un ensemble de mécanismes d'extension définis par son méta modèle : stéréotypes, valeurs marquées et contraintes. Il s'agit de l'extension d'UML par stéréotypage. Il est possible également d'étendre UML par méta modélisation.

5.1 Extension par stéréotypage

UML définit un ensemble d'éléments de modélisation dont chacun est représenté par une méta classe au niveau de son méta modèle. Il offre également des mécanismes d'extension de ces éléments: les stéréotypes (*stereotypes*), les valeurs marquées (*tagged values*) et les contraintes (*constraints*), permettent l'extension et la spécialisation des classes de concepts et relations standards d'UML. Les stéréotypes permettent d'ajouter de nouvelles classes d'éléments de modélisation au méta modèle d'UML., par dérivation des métas classes existantes, en plus du noyau prédéfini par UML. Les valeurs marquées étendent les attributs des classes d'éléments du méta modèle et les contraintes sont des relations sémantiques entre éléments de modélisation qui définissent des conditions que doit vérifier le système.

Ces mécanismes ont été exploités dans le cadre de plusieurs travaux, visant à étendre UML pour intégrer les aspects. [30]

5.2 Extension par méta modélisation

Une autre possibilité d'extension d'UML consiste à étendre son méta modèle en y ajoutant de nouveaux éléments originaux de modélisation et donc de nouveaux concepts et relations. Elle peut être complémentaire de l'utilisation des mécanismes d'extension. En effet, l'usage exclusif des mécanismes d'extension nous semble insuffisant parce qu'il repose trop fortement sur la définition de stéréotype: introduire un concept ou une relation propre à l'approche Aspect en se limitant à la définition d'un stéréotype consiste plus à se rapprocher superficiellement et artificiellement d'un élément de modélisation déjà existant dans UML que de définir un nouvel élément et de rechercher sa place dans le noyau d'UML déjà constitué.

La méta modélisation offre de plus amples possibilités de structuration dans la définition de nouveaux éléments de modélisation tout en permettant de les mettre en relation entre eux, mais également avec les concepts standards déjà présent dans le noyau d'UML .En particulier, l'usage de la généralisation permet de dégager des concepts et relations suffisamment abstraits pour pouvoir rapprocher de manière significative des modèles ou langages basés sur des concepts voisins mais néanmoins différents.

On peut noter, enfin, que l'approche par méta modélisation présente un inconvénient principal par rapport à l'approche par stéréotypage, ou profile UML .Il est difficile, en effet, d'utiliser un outil de modélisation standard UML déjà existant, pour exprimer des modèles instances du nouveau méta modèle résultat de l'extension. Nous tenons pourtant à adopter cette approche par méta modélisation dans la définition des nouveaux concepts et relations de notre langage de modélisation [30].

6 Travaux sur la POA

Nous présentant ici quelques travaux sur la POA dans des différents domaines :

Berkane [46] a exposé un modèle de transformation qui permet de traduire des instances des patterns de l'objet vers des instances en aspect. Pour cela, il a implémenté des règles qui facilitent ce passage, pour permettre la génération de code aspect, ainsi la rétro-ingénierie, pour extraire le logique métier, C'est une approche constitue en un ensemble d'algorithmes destiné à traduire des patterns.

Bouanaka [47] a présenté une approche pour l'intégration du paradigme orienté aspect dans le modèle cas d'utilisation. Il a ajouté de nouvelles notations pour le package cas

d'utilisation pour capturer essentiellement les exigences dites non fonctionnelle du serveur d'application. Pour cela, il a introduit quatre concepts pour le modèle cas d'utilisation d'ULM : *advice case system*, *advice case application*, *use case crosscut* et enfin *point cut association.*

7 Conclusion

Le développement applicatif et les méthodologies liées ont fortement évoluées tout au long de l'histoire de l'informatique. Le but recherché étant de pouvoir produire des logiciels répondant le mieux possible à la demande du client en moins de temps possible.

La programmation orientée objet a su introduire de nouveaux concepts intéressants avec ce même objectif de produire des produits de qualité. Toutefois, nous pouvons encore trouver certaines limites à ces concepts que l'on aimerait pouvoir surpasser. Ce sont ces limites que la POA se propose de résoudre en introduisant les notions de séparation des préoccupations. [39]

Nous avons présenté dans ce chapitre les principes concepts de la programmation orientée aspect, les points de jonction, les coupes, les codes advice, et le mécanisme d'introduction. Nous avons aussi présenté les concepts de langage AspectJ, support de notre travail de recherche.

Comme notre travail consiste a propose une approche intégrant des techniques de qualité de service des logiciels dans un seul aspect, nous avons choisi UML comme un standard de modélisation pour lui étendre sont méta modèle, pour cela nous allons motiver ce choix et présenter les mécanismes d'extension d'UML pour modéliser les aspects.

Chapitre III

Une approche combinant le design par contrat et le test de couverture dans la programmation orientée aspect

1 Introduction

Notre objectif principal est de définir une approche intégrant la technique de design par contrat et le test de couverture qui sont complémentaires pour permettre une meilleure qualité de service des logiciels dans le domaine de la programmation orientée aspect afin de bénéficier des avantages suivantes : la localisation, la réutilisation, la composition et de réduire le temps, l'effort ainsi le coût dans le développement et la maintenance des systèmes orientés aspects. Ces avantages sont considérés comme le but majeur de notre travail.

Nous proposons deux approches : la première approche consiste à implémenter chaque technique à part dans un aspect pour permettre d'assurer le passage de la Programmation Orientée Object vers la Programmation Orientée Aspect. La deuxième est de fusionner la technique de design par contrat et le test de couverture dans un seul aspect.

Dans ce chapitre nous présentons l'approche globale de notre travail. Puis nous présentons la stratégie de transformation d'OCL contrats par AspectJ. Cette transformation peut apparaître comme une génération de code. Et nous donnons les avantages et les inconvénients de cette approche. Nous terminons ce chapitre par la présentation de notre deuxième approche d'intégration de design par contrat et test de couverture dans un seul aspect par AspectJ et nous donnons l'algorithme d'intégration et les avantages de cette intégration.

2. Aperçu sur l'approche globale

Dans cette section, nous allons présenter un aperçu sur notre stratégie de transformation d'une manière globale et abstraite. Puis nous présentons par la suite, les règles de transformation.

Les travaux sur les différentes techniques de qualité de service du logiciel, présenter dans le Chapitre I, insiste sur la résolution des problèmes et les défauts de chaque technique à part. Donne des solutions et des approches d'utilisation de chaque technique avec l'utilisation de la POA.

Pour bénéficier de tous les avantages d'utilisation de techniques de design par contrat et test de couverture, nous proposons deux approches : implémenter chaque technique à part dans un aspect pour permettre d'assurer le passage de la POO vers la POA, cette première approche est basée sur un modèle étoile M=(C,P), c'est-à-dire l'application est au centre (C) et les pétales (P) sont les aspects des techniques, pour cela, notre modèle de transformation vise à définir des règles de transformation qui assure le passage d'un diagramme de classe UML/OCL dans le paradigme objet vers un diagramme de classe UML dans le paradigme aspect.

La transformation des techniques de design par contrat et test de couverture revient en général à :

- La transformation de Précondition de méthode d'OCL dans le diagramme de classe dans le modèle conceptuel de la programmation orientée objet à un aspect dans le diagramme de classe dans le modèle conceptuel de la POA.
- La transformation de Postcondition de méthode d'OCL dans le diagramme de classe dans le modèle conceptuel de la programmation orientée objet à un aspect dans le diagramme de classe dans le modèle conceptuel de la POA.
- La transformation d'invariant de classe d'OCL dans le diagramme de classe dans le modèle conceptuel de la POA à un aspect dans le diagramme de classe dans le modèle conceptuel de la programmation orientée aspect.

- La transformation de méthode de test de couverture dans le modèle physique de la programmation orientée objet à un aspect dans le diagramme de classe dans le modèle conceptuel de la programmation orientée aspect.

Dans notre seconde approche, nous proposons de transformer l'application avec ses aspects des techniques d'un modèle étoile à un modèle arbre M=(R, F), c'est-à-dire fusionner la technique de design par contrat et le test de couverture dans un seul aspect présente la racine d'arbre(R).

Pour réaliser cette transformation nous avons choisi le langage UML, comme représentation du modèle conceptuel orienté objet (est exactement le diagramme de classes), car il permet d'exprimer et d'élaborer des modèles objets, indépendamment de tout langage de programmation. De plus il permet un gain de précision, de stabilité, de plus il encourage l'utilisation d'outils.

Nous avons sélectionné UML pour AspectJ, pour présenter le modèle conceptuel orienté aspect. Nous avons opté la solution Suzuki [44], car cette solution représente l'aspect par les stéréotypes. Les stéréotypes permettent d'utiliser un outil de modélisation standard UML déjà existant, pour exprimer des modèles.
Le langage AspectJ est le langage utilisé pour représenter le modèle physique orienté aspect. Ce langage supporte, plus de types de points de jonction, plus de type de 'code advice',

3. Transformation de design par contrat et test de couverture dans des aspects par AspectJ

Toute application doit être testée afin de garantir à l'utilisateur final une qualité de service qui soit satisfaisante, la POA nous permet d'instrumenter le code pour contrôler son exécution.

Dans notre première approche, nous proposons modéliser chaque technique sous forme d'un aspect ou plusieurs aspects et les greffer dans l'application. Pour implémenter les aspects des techniques nous proposons que chaque équipe implémente une technique :

-Une équipe implémente la technique de design par contrats en trois aspects : Aspect_Precondition, Aspect_Postcondition, Aspect_Invariant ; qui sont greffés dans l'application.

- Une autre équipe implémente la technique de Test de couverture en un autre aspect. Apres l'implémentation des aspects nous les greffons dans l'application .L'application devient comme montre la Figure III.01 :

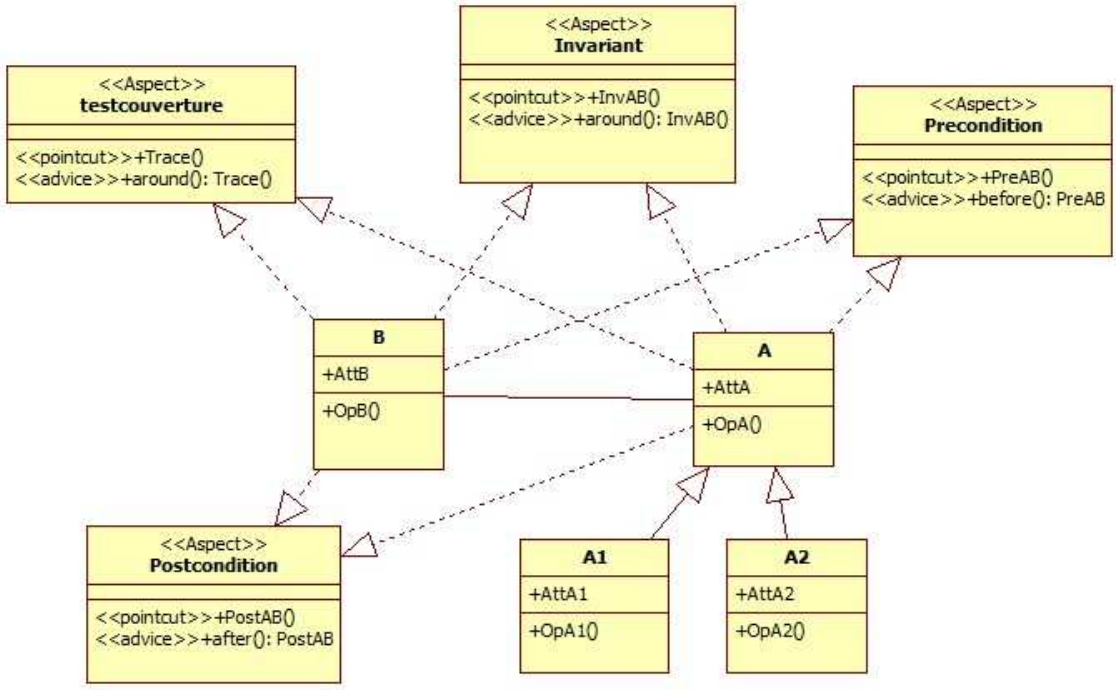


Figure III.01 *Les aspects des techniques de gestion de QOS dans l'application*

Nous trouvons que l'application est au centre et ces pétales sont les aspects des techniques.

- Aspect Precondition : spécifie la/les condition(s) nécessaire(s) pour qu'un client soit autorisé à appeler cette méthode ;
- Aspect Postcondition : exprime la/les propriété(s) qui doi(ven)t être vérifiée(s) lorsque la méthode retourne (termine), si sa pré-condition était satisfaite au moment de l'appel ;

- Aspect Invariant : spécifie des propriétés vraies pour l'ensemble des instances de la classe ;
- Aspect Test de couverture et Aspect_Test de Non régression : regroupe les différentes méthodes permettant de vérifier que chaque partie d'une application est couverte par une campagne de test. Un outil de test de couverture est composé de deux éléments : l'enregistreur et le comparateur. L'enregistreur est chargé d'analyser l'exécution de l'application et d'enregistrer les méthodes appelées et les attributs accédés. Le comparateur confronte ces enregistrements avec le code source afin d'identifier les méthodes non appelées et les attributs non accèdes. L'enregistreur est implémenté grâce aux techniques de la programmation orientée aspect (POA) pour permettent d'intercepter les appels aux méthodes et les accès aux attributs. La programmation orientée aspect (POA) est particulièrement performante pour développer l'enregistreur puisqu'elle permet d'intercepter sans difficulté les appels aux méthodes afin de réaliser divers traitements.

3.1 Stratégie de transformation d'OCL contrats en AspectJ

Le principe de cette transformation vise à définir des règles de transformation qui assure le passage d'un diagramme de classe au niveau orienté objet vers un diagramme de classe au niveau orienté aspect. Nous allons considérer que ce diagramme de classe qui sera transformé, est présenté comme un modèle conceptuel orienté objet. Après l'application nos règles de transformation, nous obtenons un modèle conceptuel orienté aspect et nous générons un code aspect en AspectJ (au lieu de générer un code JAVA correspondant au diagramme de classe et JML pour les contraintes OCL).

Pour réaliser cette transformation, nous allons suivre certaines règles, qui seront présentées par la suite. La stratégie de traduction d'OCL contrats utilise AspectJ est représentée dans la Figure III.02 :

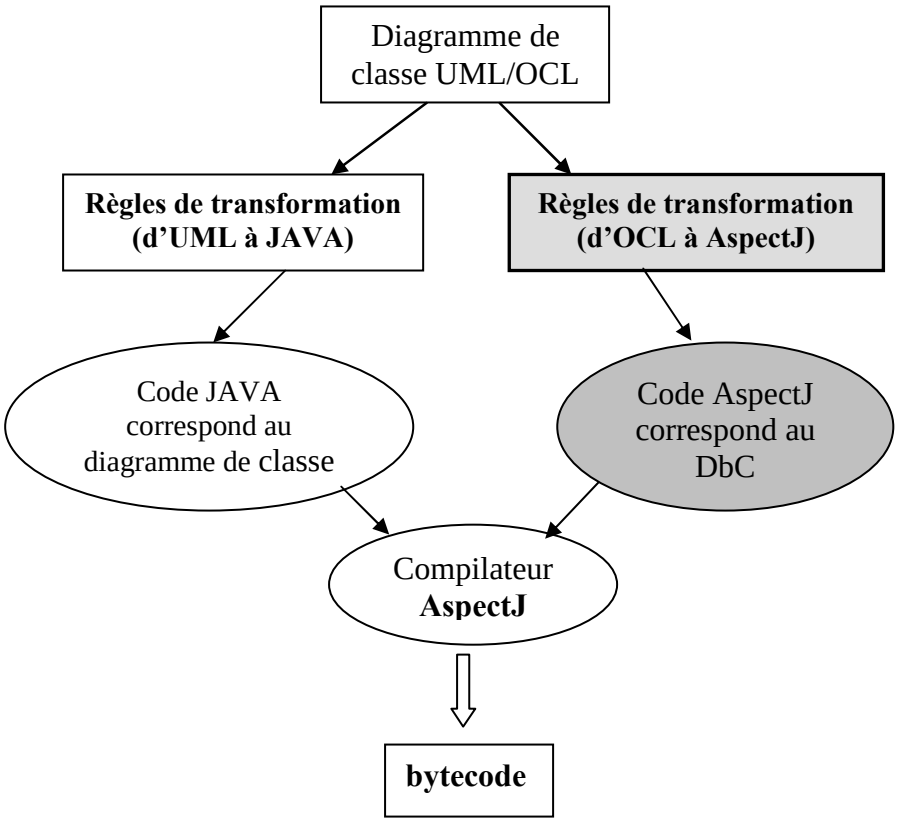


Figure III.2 *Une vue de stratégie de traduction d'OCL contrat utilisant AspectJ*

Le but de notre travail est de définir une ensemble de règles de transformation entre les diagrammes de classes UML intégrant OCL contrats et AspectJ. La principale force de cette stratégie consiste à fournir un moyen d'établir la cohérence entre le modèle de conception et d'implémentation.

Dans cette section, nous présentons une transformation d'OCL contrat en un aspect. Nous concentrons sur la précondition , postcondition de méthode et invariant de classe pour la conception par contrat. Pour les règles de transformation nous considérons l'annotation d'OCL suivante :

Context C

inv: θ

Context C : : M ()

pre : γ

post : α

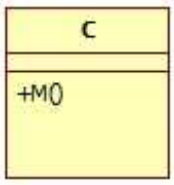
Cette annotation fournit une classe C avec une méthode M avec une précondition γ ; postcondition α, invariant θ.

Dans la conception par contrat et OCL, γ, α, θ sont des expressions booléenne .Les OCL contrats sont des expressions booléennes construites en utilisant les opérations booléennes *and*, *or* et *not*. L'expression p and q, p or q, not p sont correspondant aux expressions AspectJ suivantes p & &q, ¬ p | | q,! p

a) Règle 01 : Transformation de Précondition de méthode

En Design par contrat, les préconditions doivent être remplies avant que le code de la méthode est exécuté, si elle n'est pas satisfaite, elle conduit à une séquence irréalisable.
La précondition de design par contrat est représentée par UML/OCL comme suit:

Context C : : M ()
pre : γ



Nous pouvons transformer cette condition sur Aspect. Le modèle conceptuel qui présente la précondition de la méthode M dans un aspect est le suivant :

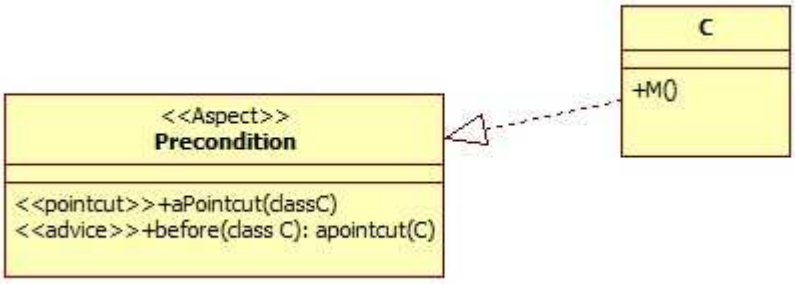


Figure III.03 *Modèle conceptuel représente les preconditions pour AspectJ*

Nous pouvons traduire cette condition sur AspectJ. Le code AspectJ qui vérifie la précondition de la méthode M est le suivant :

```
before ():
 execution (void M ()) && within(C) {
  if (! γ) {
 throw new   PreconditionViolationException();
              }
```

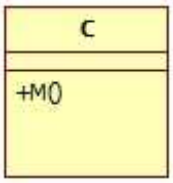
Cet extrait de code, une partie d'un aspect, déclare qu'un advice est invoqué avant chaque exécution de la méthode M de la classe C ou de l'un de ses sous-classes. Le code advice vérifie la précondition , et lance une exception si elle est violée.

La première ligne indique les méthodes qui seront affectés. Dans ce cas, l'exécution de méthode M dans la classe C est sans paramètres.

b) Règle 02 : Transformation de Postcondition de méthode

Les postconditions sont des propriétés dans les OCL contrats qui doivent être vrais après l'exécution d'une méthode, sinon ils conduisent à la mise en œuvre d'erreur dans la méthode.

Context C : : M ()
post : α



Nous pouvons transformer cette condition sur Aspect. Le modèle conceptuel qui présente la postcondition de la méthode M dans un aspect est le suivant :

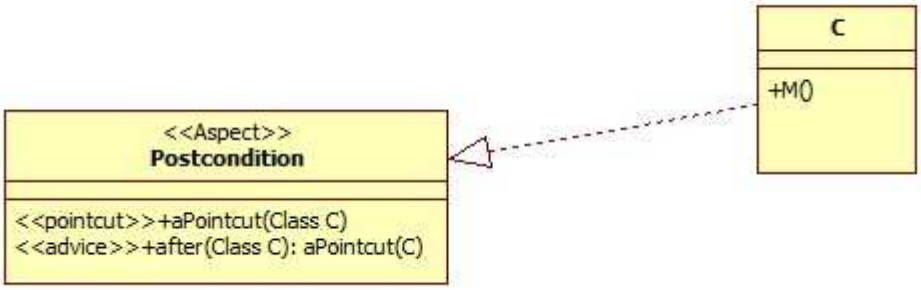


Figure III.4 *Modèle conceptuel représente les postconditions pour AspectJ*

Afin de traduire les postcondition OCL, nous avons utilisé deux types d'advice: *after returning* et *after throwing* . *after returning* est appliqué lorsque la méthode retourne normalement sans lancer aucune exception. D'autre part, lorsque l'exécution de la méthode se termine par un lancement d'une exception, l'advice *after throwing* est appelé pour ajouter le code après l'exécution de la méthode.
La méthode insérée à vérifier la postcondition est le suivant :

```
public Boolean C.check_postcondition_M()
        {
return α ||super.check_postcondition_M();
        }
 }
```

Le code AspectJ, qui met en œuvre les advices de postcondition de ceci est la suivante :

```
after(C current)returning(Object o) :
execution(!static * C.*(..)) && within(C) && this(current){
/*Il vérifie la postcondition, s'il détecte toute violation
  de postcondition, alors qu'il jette un ProconditionViolationException,
  sinon il retourne normalement.*/
if (!current.check_Postcondition_M()){
throw new ProconditionViolationException();
        }
}
after(C current)throwing(Throwable throwable) :
execution(!static * C.*(..))&&within(C) && this(current){
/*Ce conseil vérifie quel type d'exception a été jeté*/
 if(throwable instanceof σ){
   throw(σ)throwable;
                          }
if(throwable instanceof π){
 if (!current.check_Postcondition_M()){
throw new PostconditionViolationException();
}
else{
throw(π)throwable;
                  }
}
}
```

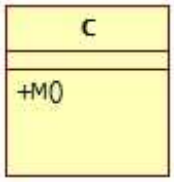
Les advices après le retour sont exécutés. Il vérifie la postcondition et si elle détecte une violation de postcondition . Il exécute un PostconditionViolationException (), sinon retourne normalement.

c) Règle 03 : Transformation d'invariant de classe

Les invariants de classe expriment les propriétés d'une classe qui doivent être préservées par toutes les routines de la classe.

Invariant de classe est exprimé comme suit:

Context C :
inv: θ



Le mot-clé *context* indique le cadre de l'invariant, qui est la classe C dans le cas présent, ***inv*** est indique le type de la contrainte qui est un invariant. L'invariant θ lui-même est une expression booléenne.

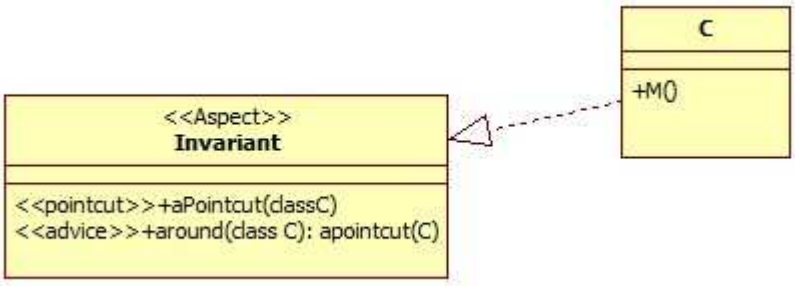


Figure III.5 *Modèle conceptuel représente les invariants pour AspectJ*

Cet invariant est invoquée par *before* et *after advice*, il est traduit à AspectJ dans le code suivant:

```
before(C current) :
 execution(!static * C.*(..)) && within(C) && this(current){
    if (!current.check_Invariant_C()){
  throw new InvariantViolationException();
}
}
```

```
after(C current)returning(Object o) :
execution(!static * C.*(..)) && within(C) && this(current){
/*Il vérifie l'invariant, s'il détecte toute violation d'invariant,
alors qu'il jette un InvariantViolationException, sinon il retourne
normalement.*/
if (!current.check_Invariant_C ()){
 throw new InvariantViolationException();
        }
}
after(C current)throwing(Throwable throwable) :
execution(!static * C.*(..))&&within(C) && this(current){
/*Ce conseil vérifie quel type d'exception a été jeté*/
 if(throwable instanceof σ){
   throw(σ)throwable;
                          }
if(throwable instanceof π){
 if (!current.check_Invariant_C ()){
throw new InvariantViolationException();
}
else{
throw(π)throwable;
                  }
}
}
```

Comme pour les postconditions, nous utilisons *after returning advice* et *after throwing advice* pour vérifier l'invariant.π, σ présente les exceptions Java, les exception AspectJ respectivement.

3.2 Stratégie de transformation de méthode de test de couverture par AspectJ

Le logiciel « zéro défaut » n'existe pas. La présence de fautes logicielles systématiques, introduites à la conception d'un dispositif programmé, doit donc être considérée avec beaucoup d'attention, en particulier lorsque les conséquences de ces fautes peuvent influer sur la sécurité d'un dispositif complexe.

Dans cette section nous proposons une stratégie de mise en œuvre d'un aspect de tests de couverture, qui est le principal composant de la vérification d'un logiciel. Pour détecter un maximum de fautes.

L'un des exemples les plus couramment utilisé pour introduire le test de couverture est la gestion des traces. Effectivement, si nous voulons afficher quelque chose à chaque appel de méthode, ou d'instance d'objet, vous serez obligés d'ajouter une ligne de code avant chacun de ces points du programme. Il y a donc recopie fréquente de codes dans l'ensemble du programme pour ajouter la fonctionnalisée de gestion de traces.

La POA est là encore capable de résoudre ce problème, en ajoutant un aspect qui capture les appels de méthodes dont le code advice serait l'affichage de la trace. Ceci est de plus très pratique, car facilement et rapidement implémentable sur un programme déjà important, mais aussi facilement supprimable.

Le modèle conceptuel représente la gestion de trace que nous pouvons greffer dans n'importe quelle application sans aucune modification dans l'application ou dans l'aspect lui-même est le suivant :

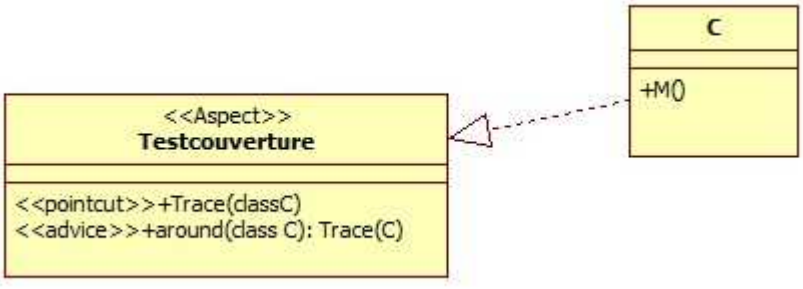


Figure III.06 *Modèle conceptuel représente la gestion de trace pour AspectJ*

Ce modèle conceptuel de gestion de trace est traduit à AspectJ dans le code suivant:

```
public aspect testcouverture {
pointcut Trace ():
call( public void *.*(..) )
void around():Trace () {
String methodName = thisJoinPoint.getSignature().getName();
        Object[] args = thisJoinPoint.getArgs();
        Object caller = thisJoinPoint.getThis();
        Object callee = thisJoinPoint.getTarget()
 System.out.println("-> Début méthode"+methodName); System.out.println("-
>"+args.length+"parameter(s)");
For (int i=0;i<args.length;i++)
        System.out.println(args[i]+" ");
        System.out.println();
        System.out.println("->"+caller+" vers "+callee);
        Object ret=proceed ();
        System.out.println("-> Fin methode"+methodName);
        Return ret;

        }
```

Pour montrer ce que peut être une stratégie de test de validation, une stratégie spécifique est à élaborer pour chaque contexte spécifique, et pour un contexte donné. La liste ci-dessous fournit des exemples d'étapes typiques. Ces étapes sont liées aux types de tests ainsi qu'aux contraintes de mise en œuvre des tests (banc de test,
environnement…) :

_ Une étape de test détaillé pour chaque fonction en nominal ;

_ Une étape de test détaillé pour chaque fonction aux limites ;

_ Une étape de test détaillé pour chaque fonction hors limites ;

_ Une étape de test de performance ;

_ Une étape vérifiant le comportement en cas de défaillance ;

_ Une étape consacrée au paramétrage ;

_ Des étapes avec des entrées simulées et des étapes avec des entrées réelles du milieu opérationnel ;

La stratégie doit s'intégrer dans la stratégie d'ensemble de vérification du logiciel.

En pratique, il arrive que les essais ne couvrent pas totalement les objectifs fixés,

- parce que certains défauts ou certains modes de défaillance ne peuvent pas être simulés, ou
- parce que certains états ne sont jamais atteints dans l'environnement du système.

La stratégie montrer quelles sont les vérifications complémentaires qui seront menées pour que le risque de non conformité soit amené à un niveau raisonnable, par exemple, le renforcement de certains tests en phase de tests unitaires ou d'intégration.

3.3 Avantages de transformation de chaque technique dans un aspect à part

Nous avons vu beaucoup d'avantage dans cette transformation, nous pouvons citer les suivants

- ✓ L'indépendance d'implémentation des techniques assure un bon test et une bonne construction des aspects.
- ✓ L'implémentation parallèle des techniques permet de gagner beaucoup de temps.
- ✓ Chaque aspect a un ensemble des points de jonction bien précise.
- ✓ L'indépendance d'implémentation des techniques assure une mise à jour des aspects très faciles.
- ✓ L'indépendance d'implémentation des techniques permet de déminer le coût et l'effort de développement, et augmenter la Confiance des logiciels développés.
- ✓ L'indépendance d'implémentation des techniques assure une meilleure séparation des préoccupations

3.4 Inconvénients de transformation de chaque technique dans un aspect à part

- La Figure III.01, montre qu'il y a plusieurs points de jonction pour greffer les aspects sur l'application, qu'il construit un conflit des points jonction, le test de programme orientée aspect est plus difficile que les traditionnels programmes [16].

Si nous utilisons l'approche de test des programmes orienté aspect définie dans [16] pour seulement les points jonction d'appelle d'une méthode d'une classe on obtient le graphe de la Figure III.04 (Pour plus d'explication voir [16])

Mario et al [16] propose un ensemble de critères de tests de couverture pour les programme orientée aspect basés sur les interactions entre les codes advices et les méthodes. Les critères de couverture exploitent le graphe de Flux de contrôle d'Inter- procédure et l'aspect (IACFG : Inter-procedural Aspect Control Flow Graph). Un graphe représente des relations entre les codes advices des aspects et les méthodes des classes, en tenant compte de la POA et des caractéristiques uniques de langage à un niveau de faible granularité.

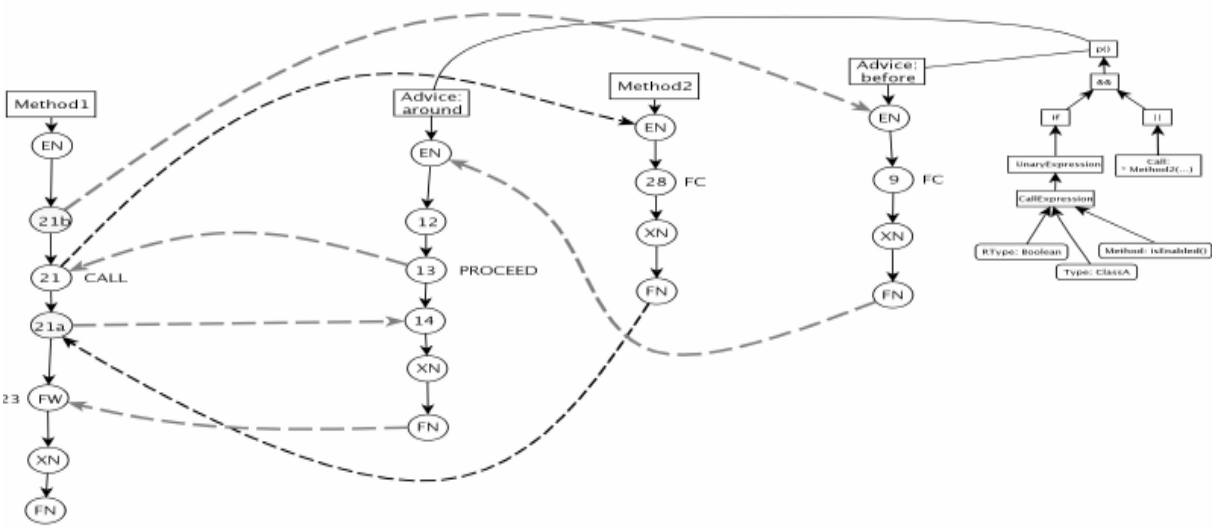


Figure III.7 : *Exemple d'IACFG de POA [16]*

Nous pouvons imaginer si nous ajoutons tous les aspects des trois techniques de notre application, le graphe est complexe et illisible.

- Si un appel d'une méthode M est un point de jonction qui trouve dans tous les coups des aspects d'application ; la modification de nom ou de signature de M nécessite de modifier les coupes de tous les aspects. Ces modifications sont d'autant plus coûteuses que la méthode est d'usage courant
- Dans le code des aspects des trois techniques greffées, nous trouvons des dispersions de code, beaucoup de redondance (redondance des méthodes, de test …). Cette dispersion est très claire sur les exemples des aspects de chaque technique écrite dans [17].
- Si nous modifions le nom d'une classe ou d'une méthode, et nous ne modifions pas les point de coupure des aspects qu'ils contiennent cette classe ou méthode, après compilation de logiciel, le compilateur ne signale aucune erreur et le logiciel s'exécutera normalement son des bogues. Mais les aspects n'exécuteront jamais parce qu'ils ne capturant jamais les point de jonction.
- Si nous greffons les aspects dans l'application en même temps, l'exécution de l'application donne chaque fois un résultat : les aspects de conception par contrat puis l'aspect de test de couverture ou l'inverse c'est-à-dire la séquence d'exécution est indéterministe,

4. Intégration de design par contrat et test de couverture dans un seul aspect par AspectJ

Nous proposons une deuxième approche qu'intègre ces techniques en un seul aspect : une seule technique peut ne pas suffisons pour garantie la qualité d'un logiciel car chacune concentre sur un ensemble des exigences. Par conséquent, une combinaison de différentes techniques peut êtres utile mais elle doit êtres gérée avec prudence.

Nous choisissons la conception par contrat et le test de couverture pour assurer que toutes les méthodes sont appelées et toutes les variables sont utilisées pour cela nous implémentant un aspect qui donne comme résultat un fichier de trace.

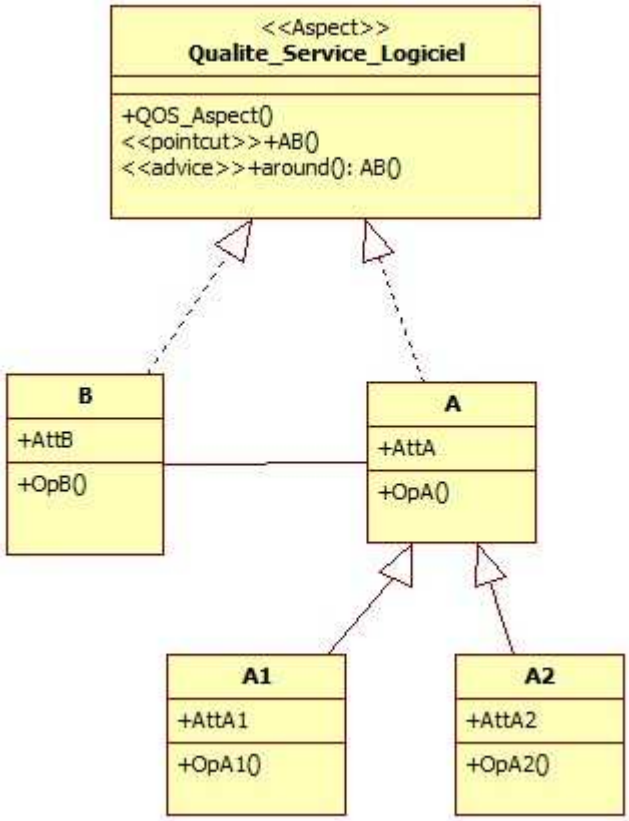


Figure III.8 *Modèle arbre pour la gestion de qualité de service de logiciel*

Dans notre seconde approche, nous proposons de transformer l'application avec ses aspects des techniques d'un modèle étoile à un modèle arbre M=(R, F), c'est-à-dire fusionner la technique de design par contrat et le test de couverture dans un seul aspect présente la racine d'arbre(R).

Un aspect regroupe les pointcuts et les advices. L'aspect ressemble à une classe: un aspect peut contenir des méthodes et des champs. Notre aspect de qualité de service contient un ensemble de pointcut et un code advices de post et précondition et invariant et méthode de trace. Nous donnons dans la suite l'algorithme d'intégration de design par contrat et test de couverture dans un seul aspect.

4.1 Stratégie d'intégration de design par contrat et test de couverture dans un seul aspect par AspectJ

La couverture de code est une mesure qui décrit le degré auquel le code source d'un programme a été testé. Ainsi c'est l'exécution de tous les tests (unitaires, fonctionnels, etc...) qui va permettre de déterminer notre mesure de couverture.

Cependant, notre stratégies de transformation, traduit un modèle source .Ce modèle doit être en orienté objet. Il est représenté soit par UML pour le modèle conceptuel, soit par Java pour le modèle physique vers un modèle cible (Ce dernier doit être en orienté aspect. Il est décrit soit par UML/AspectJ pour le modèle conceptuel, soit par AspectJ pour le modèle physique).

4.2 Algorithme d'intégration de design par contrat et test de couverture dans un seul aspect

Pour intégrer la technique de design par contrat et le test de couverture dans un seul aspect nous proposons l'algorithme suivant qui demande en entrée l'ensemble de point de jonction et donne on sortie un fichier de trace.

Algorithme intégration de design par contrat et test de couverture

Déclaration

- ✓ Déclaration d'une méthode dans l'aspect qui permet d'écrire dans un fichier qui affiche la trace d'exécution de l'application sous forme d'un fichier contenant les informations suivantes :
  - Nom de classe
  - Nom d'opération
  - Nom de constructeur
  - Arguments de la méthode
  - Type d'appel

On ajoute les champ suivants préconditions , postconditions et invariants pour vérifier que chaque appel d'une méthode respecte les contraintes de contrat.

Début

- ✓ Pour chaque appel nous interceptons ces informations et vérifier les préconditions de la méthode :

Si (**les préconditions (γ) sont non vérifiées**) **Alors**

-Mise à jour de fichier de trace, ajouter les informations suivantes :

nom de classe, mon d'opération ou constructeur, les arguments, la précondition non vérifier (faux)

-Lever une exception et écrire un message d'erreur.

Sinon (** les précondions(γ) sont bien vérifiées**)

-Sauvegarder les attributs nécessaires pour la vérification des Postconditions et des invariants (pour résoudre le problème vieilles valeurs Dans postcondition [01]).

-Retourner ver l'application pour cantinier l'exécution par la méthode proceed() de l'aspect dans AspectJ.

-A la fin d'exécution d'une méthode en retourner à l'aspect pour vérifier les Postconditions et les invariants.

Si (**les postconditions(* α) *vérifiées**) **Alors**

-Vérifier les invariants de la classe.

Si (**invariant* (θ) *vérifies**) **Alors**

- Mise à jour de fichier de trace, ajouter les informations suivantes :nom de classe, mon d'opération ou constructeur, les arguments, la précondition vérifier, postcondition vérifier, invariant vérifier

-Retourner ver l'application par la méthode proceed() de l'aspect dans AspectJ.

Sinon

- Mise à jour de fichier de trace, ajouter les informations suivantes : nom de classe, mon d'opération ou constructeur, les arguments, précondition vérifier, postcondition vérifier, l'invariant condition non vérifier)

-Lever une exception et écrire une message d'erreurs.

Sinon

- Mise à jour de fichier de trace, ajouter les informations suivantes : nom de classe, mon d'opération ou constructeur, les arguments, précondition vérifier, postcondition non vérifier)
-Lever une exception et écrire un message d'erreurs.

Fin

La traduction de cet algorithme en AspectJ nous donnons le modèle d'Aspect général d'intégration de DbC et test de couverture qui peut utiliser pour les applications orientés objet nous modifier seulement γ, α, θ pour les précondition ,postcondition et invariant des classe respectivement.

4.3 Avantages d'Intégration de design par contrat et test de couverture dans un seul aspect

Nous avons vu plusieurs bénéfices dans l'intégration de design par contrat et test de couverture dans un seul aspect. La motivation de notre travail est basée sur plusieurs avantages :

- Un seul aspect greffé sur l'application permet d'éliminer carrément le problème de conflit des aspects parce que nous présentons un seul aspect qui sera greffé dans tous les points de jonction.
- Elle peut réduire considérablement la taille d'un programme Java sans perte de performance, et simplifie la conception d'autant.
- L'implémentation d'un seul aspect permet de gagner beaucoup de temps.
- Une meilleure documentation du code et réutilisation d'aspect. Les contrats sont spécifiés clairement dans le code des composants et donc facilement lisibles par les développeurs. La réutilisation est facilitée car les contraintes liées à une classe sont exposées de manière explicite.
- Nous avons un seul ensemble des points de jonction bien précis. Cette intégration évite le problème de redondance des points de jonction.
- Particulièrement utile pour déboguer de grands projets.

- Augmentation de la qualité du code car les concepteurs et les programmeurs peuvent se concentrer sur le code métier. Les contraintes des contrats sont définies dans des modèles réutilisables.
- Permet de trouver les erreurs facilement : l'ajout des champ suivant préconditions , postconditions et invariants dans le fichier de trace pour vérifier que chaque appel d'une méthode est bien effectue permet de trouver la cause des erreurs facilement.
- Augmentation de la réutilisation du code: dans les conditions actuelles, un module implémente de multiples exigences. D'autres client nécessitant des fonctionnalités similaires pourraient ne pas pouvoir réutiliser le module tel quel, mais avec ces nouveaux concepts des le départ la réutilisation des différents aspects du système est permises
- Meilleure efficacité des tests et du débogage. Notre aspect de qualité de service permet de définir de manière beaucoup plus stricte les conditions d'exécution des différentes classes composant une application.
- Une mise à jour de cet aspect très facile et maintenance facile: les différentes contraintes de design par contrat d'une application deviennent facilement modifiables dans la conception et l'implémentation. Il en résulte une correspondance claire entre les contraintes et leurs implémentations.
- Traçage facile : l'ajout de méthode de traçage dans l'aspect de qualité de service permet de surveiller l'exécution de l'application et la vérification des contraintes de contrat.

5. Conclusion

Nous avons présenté dans ce chapitre une approche pour l'intégration de design par contrat et test de couverture par la programmation orientée aspect.

Nous avons défini des règles et des modèles de transformations, nous avons insisté sur la séparation des exigences et la réutilisation.

Nous concluons que par cette intégration des techniques de gestion de qualité de service nous pouvons augmenter le pourcentage de fiabilité et de performance des logiciels, par un meilleur test, parce que il permet de trouver plus des erreurs; on peut couvrir tout le programme avec des post et pré condition erroné ou un programme avec des pré et post condition correct et bien vérifier mais non utiliser dans le programme ou ne l'accède jamais.

On va minimiser le temps de test et l'effort ainsi le coût dans le développement des systèmes orientés aspects en comparaison avec l'utilisation de chaque technique appart.

Notre but est de faciliter la maintenabilité, la traçabilité, la réutilisation et l'évolution des logiciels. Nous motivons notre travail d'un point de vue pratique, par une étude de cas sur le chapitre suivant.

Chapitre IV

Quelque aspect d'Implémentation et étude de cas

1. Introduction

La représentation théorique de notre approche a consisté à assurer le passage des contraintes OCL de l'orientée objet vers l'orientée aspect. Le principe de cette transformation a été exposé dans le chapitre précédent. Nous avons défini un ensemble de règles pour effectuer ce passage. De plus, nous avons présenté les différentes situations de transformation.

Afin de valider nos approches, nous essayons d'implémenter un logiciel de Gestion de comptes bancaire avec nos aspects de gestion de qualité de service sous forme d'étoile et sous forme d'arbre. La première approche est l'implémentation de chaque aspect à part. La deuxième approche est de fusionne les aspects de qualité de service en un seul aspect.

Nous présenterons également, dans ce chapitre les différentes étapes que nous avons suivies pour valider nos approches.

2. Enoncé de l’étude de cas « Gestion Bancaire »

Plaçons-nous dans le contexte d’une application bancaire. Il nous faut donc gérer :

- des comptes bancaires,
- des clients,
- et des banques.

De plus, on aimerait intégrer les contraintes suivantes dans notre modèle :

- un compte doit avoir un solde toujours positif ;
- un client peut posséder plusieurs comptes ;
- une personne peut être cliente de plusieurs banques ;
- un client d’une banque possède au moins un compte dans cette banque ;
- un compte appartient forcément à un client ;
- une banque gère plusieurs comptes ;
- Une banque possède plusieurs clients.

Le but de cette étude de cas est de fournir un logiciel de gestion d'un compte bancaire .Le logiciel doit effectuer les tâches suivantes :

- Créer un compte,
- Débiter un compte,
- Créditer un compte,
- Afficher un compte,
- Consulter le solde d'un compte.

Pour cela nous avons utilisé la combinaison de l’UML (Unified Modeling Language), avec OCL est, à l'heure actuelle, probablement le meilleur moyen de développer la bonne qualité et le haut niveau de modélisation, comme il a des résultats précis, sans ambiguïté, et les modèles compatibles.

3. Diagramme de cas d'utilisation

D'après l'énoncé, nous avons fait ressortir les besoins et les événements de deux acteurs d'une banque par le moyen d'un diagramme d'utilisation :

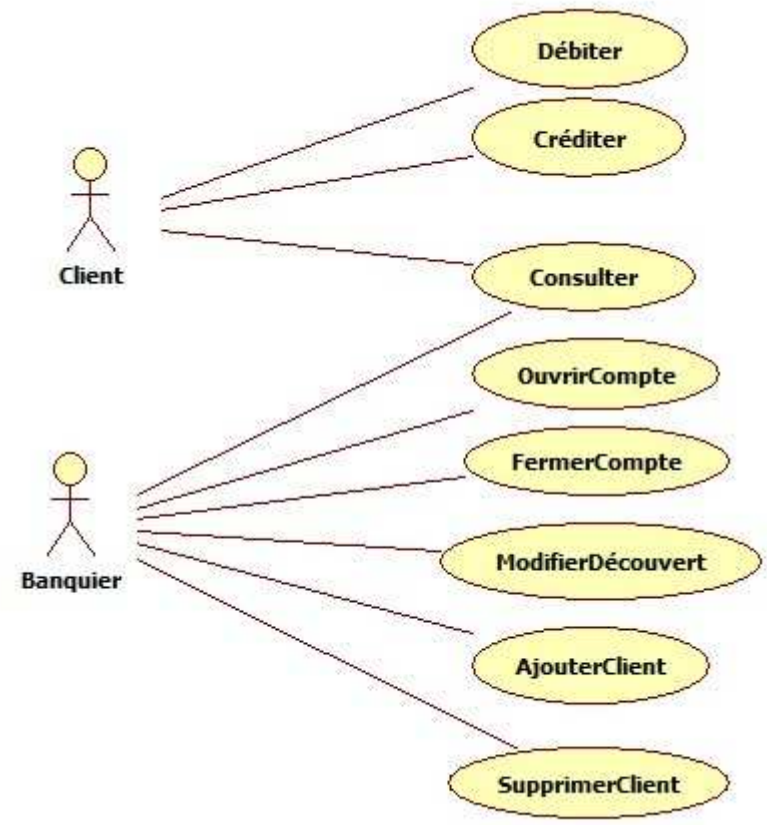


Figure IV.1 *Diagramme de cas d'utilisation de gestion bancaire*

Le diagramme d'utilisation général rédigé, à nous ensuite d'établir différents cas d'utilisation Un scénario décrit une exécution particulière d'un cas d'utilisation du début à la fin .il correspond à un enchaînement du C.U, se terminant par une fin normale ou nom (exception ou erreur).En générale, en présente les scénarios nominaux, les enchaînements alternatifs et les enchaînements d'erreurs. Dont voici un exemple : le cas « Consulter un compte ».

PROJET : Gestion de Compte
Nom Cas d'utilisation : Consulter un compte
Résumé : ce cas d'utilisation permet à n'importe quel client de l'application de consulter un compte dont il connaît le numéro à condition que celui-ci n'ait pas été clôturé.
Acteur déclencheur : Client
Acteur participant : néant

Pré conditions : L'application est exécutée et l'écran d'accueil est affiché.

Description des enchaînements :

Scénario nominal GDC01-N

1. Choisir l'action « consulter un compte »
2. Saisir le numéro de compte
3. Vérification du numéro de compte
4. Affichage du solde et du libellé
5. L'écran d'invite est affiché

Enchaînements alternatifs : GDC01-A-01

Numéro de compte invalide

L'enchaînement GDC01-A-01 démarre au point 3 du scénario nominal.

4. L'utilisateur est informé que le numéro de compte est inexistant
5. L'écran d'invite est affiché

Enchaînements alternatifs : GDC01-A-02

Numéro de compte valide mais le compte est bloqué

L'enchaînement GDC01-A-02 démarre au point 3 du scénario nominal.

4. L'utilisateur est informé le compte a été clôturé et qu'il est désormais impossible de le consulter
5. L'écran d'invite est affiché

Enchaînements d'exceptions : GDC01-E-01

Aucun numéro de compte n'est saisi

L'enchaînement GDC01-E-01 démarre au point 2 du scénario nominal.

3. L'application invite l'utilisateur à saisir de nouveau un numéro de compte
4. L'écran d'invite est affiché

4. Diagramme des classes métier UML/OCL

Pour consolider l'application, nous avons mis en œuvre le modèle d'analyse MVC (Modèle – Vue – Contrôleur). Ce modèle permet de séparer l'application en couche afin d'isoler le logique métier des classes de présentation, et d'interdire l'accès direct aux données par ces classes de présentations. Les objets « contrôleurs » jouent alors le rôle de façade vis-à-vis de la couche de présentation.

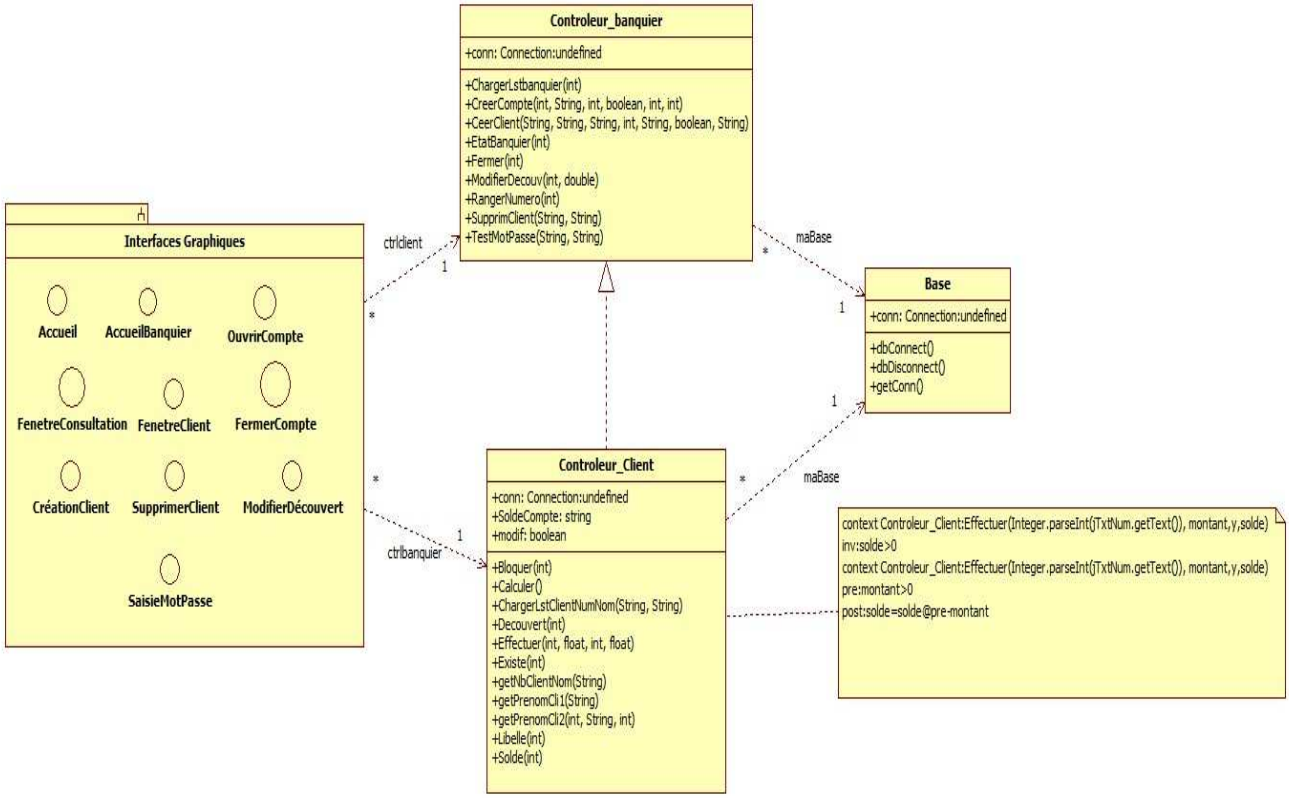


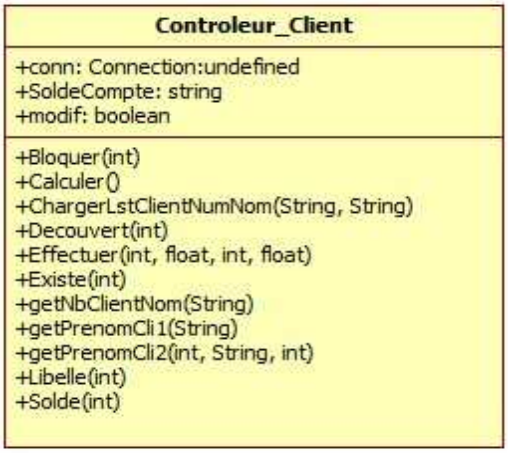
Figure IV.2 *Diagramme des classes métier UML/OCL de gestion bancaire*

5. Application de stratégie de transformation d'OCL contrats par AspectJ

Comme nous avons vu sur le chapitre III, le principe de cette transformation, est qu'à partir d'un diagramme de classe au niveau orienté objet, nous le transférons en diagramme de classes au niveau orienté aspect. Nous générons un code aspect correspondant à ce diagramme de classe. Dans cette section nous appliquons notre stratégie de transformation.

a) Règle 01 : Transformation de Préconditions des méthodes

Pour la gestion des comptes bancaire, la précondition (montant > 0) doit être vérifier avant que le code de la méthode Effectuer (Integer.parseInt(jTxtNum.getText()), montant, y, solde) est exécuté, si elle n'est pas satisfaite, elle conduit à une séquence irréalisable. Cette précondition de design par contrat est représentée par OCL comme suivantes:



Context Controleur_Client :
Effectuer(Integer.parseInt(jTxtNum
.getText()), montant, y, solde)
Pre : montant>0

Nous pouvons transformer cette condition sur Aspect. Le modèle conceptuel qui présente la précondition de la méthode Effectuer (Integer.parseInt(jTxtNum.getText()), montant, y, solde) dans un aspect est le suivant :

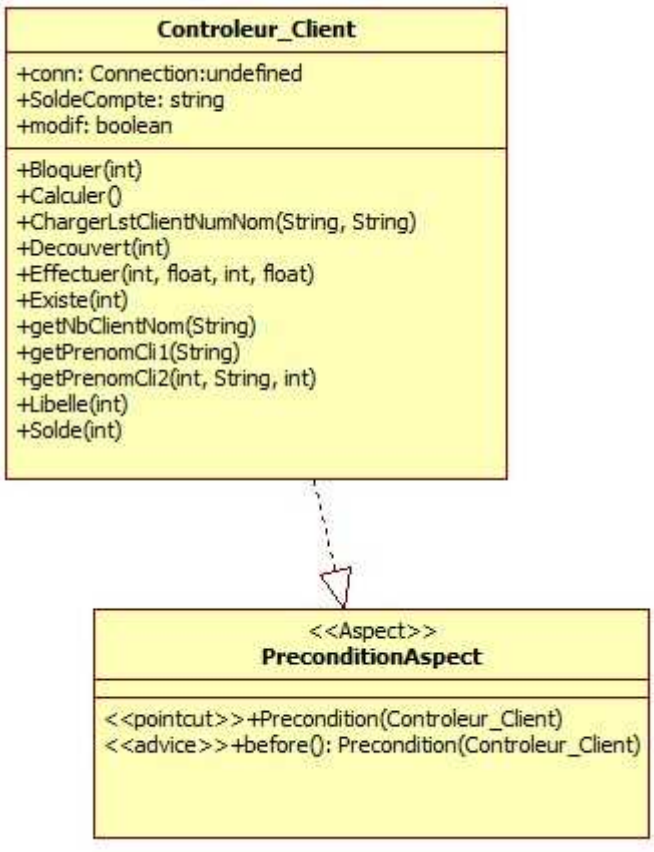


Figure IV.3 *Modèle conceptuel représente la preconditions (montant >0) par AspectJ*

Nous pouvons traduire cette condition sur AspectJ. Le code AspectJ qui vérifie la précondition de la méthode Effectuer(Integer.parseInt(jTxtNum.getText()), montant, y, solde) est le suivant :

```
before(): Precondition() {

      try {
      String methodeName= thisJoinPoint.getSignature().getName();
      Object[] args =thisJoinPoint.getArgs();
      Object caller =thisJoinPoint.getThis();
      Object callee =thisJoinPoint.getTarget();
            monFichier = new FileWriter("E:\\Trace1.txt",true);
            tampon = new BufferedWriter(monFichier);

            // Ecrit  dans Trace1.txt
            tampon.write("-> debut methode  "+methodeName);
            if (methodeName == "Effectuer"){

                  Float objectFloat=(Float)args[1];
                  Float x =objectFloat.floatValue();
                  Integer objectinteger=(Integer)args[2];
                  Integer w =objectinteger.intValue();
```

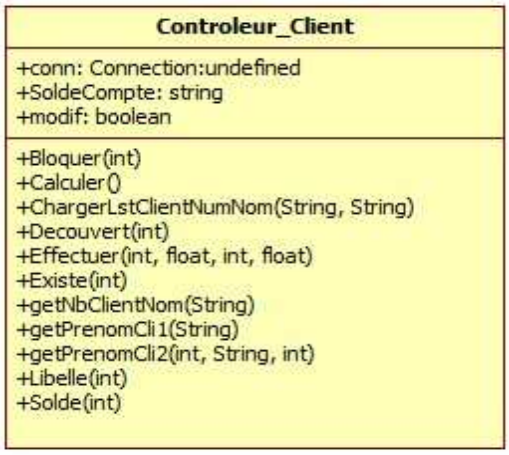
```
                if (w==1){
                x=-x;
                }
                tampon.write(x+" ");
              if (x < 0){
                tampon.write("Precondition non verifier");
                 JOptionPane.showMessageDialog(null, "Precondition non
                   verifier", "Opération  non effectuée", 0);
                tampon.newLine();
                tampon.flush();
                tampon.close();
                monFichier.close();
                 System.exit(0);
              }
          }
          tampon.write("\n");

          } catch (IOException exception) {
          exception.printStackTrace();
          } finally {
          try {
                tampon.newLine();
          tampon.flush();
          tampon.close();
          monFichier.close();
          } catch (IOException e1) {
          e1.printStackTrace();
          }
          }
}
```

Cet extrait de code, une partie d'un aspect, déclare qu'un advice est invoqué avant chaque exécution de la méthode Effectuer(Integer.parseInt(jTxtNum.getText()), montant, y, solde) de la classe Controleur_Client ou de l'un de ses sous-classes. Le code advice vérifie la précondition , et lance une exception si elle est violée.

La première ligne indique les méthodes qui seront affectés. Dans ce cas, l'exécution de méthode Effectuer(Integer.parseInt(jTxtNum.getText()), montant, y, solde) dans la classe Controleur_Client est avec paramètres.

b) Règle 02 : Transformation de Postcondition de méthode

La postcondition est une propriété dans les OCL contrats doivent être vrais après l'exécution d'une méthode dans notre application le solde après l'exécution de la méthode Effectuer(Integer.parseInt(jTxtNum.getText()), montant, y, solde) doit être égale au solde avant l'exécution de cette méthode mois le montant .Sinon il conduit à la mise en œuvre d'erreur dans la méthode et l'application doit être arrêté.



Context Controleur_Client :
Effectuer(Integer.parseInt(jTxtNum
.getText()), montant, y, solde)
Post : solde=solde@pre-montant

Nous pouvons transformer cette condition sur Aspect. Le modèle conceptuel qui présente la postcondition de la méthode Effectuer(Integer.parseInt(jTxtNum.getText()), montant, y, solde) dans un aspect est le suivant :

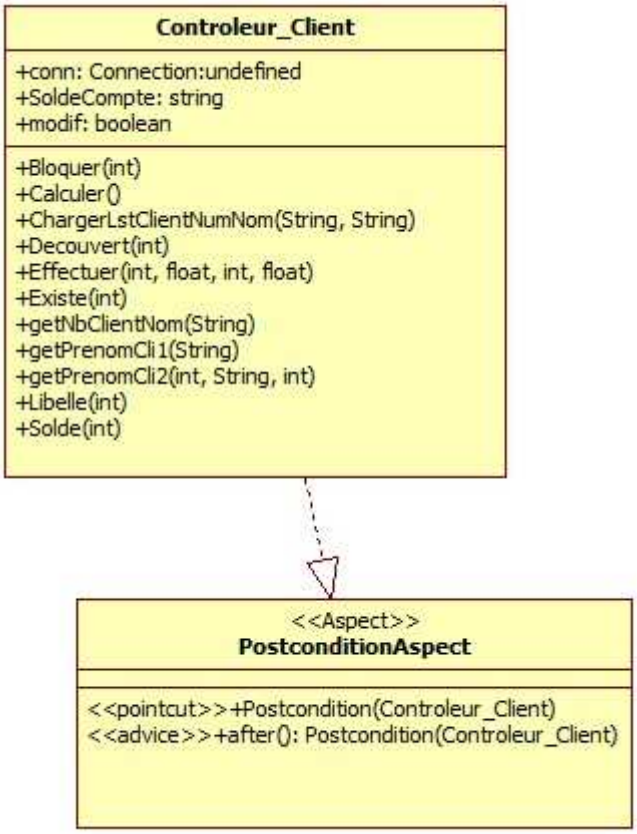


Figure IV.4 *Modèle conceptuel représente la postcondition pour la Classe Controleur_Client par AspectJ*

Nous pouvons utilisé deux types d'advice: *after returning* et *after throwing* . *after returning* est appliqué lorsque la méthode retourne normalement sans lancer aucune exception. D'autre part, lorsque l'exécution de la méthode se termine par un lancement d'une exception, l'advice *after throwing* est appelé pour ajouter le code après l'exécution de la méthode.
La méthode insérée à vérifier la postcondition est le suivant :

```
public Boolean Controleur_Client.check_postcondition_
Effectuer(Integer.parseInt(jTxtNum.getText()), montant, y, solde)
        {
return solde= =solde@pre-montant ||super.check_postcondition_
Effectuer(Integer.parseInt(jTxtNum.getText()), montant, y, solde) ;
        }
```

Le code AspectJ, qui met en œuvre les advices de postcondition de ceci est la suivante :

```
public aspect PostconditionAspect {
       int i=1;
          FileWriter monFichier = null;
            BufferedWriter tampon = null;
          /**
           * Cette coupe est tres detaillee puisque seuls les appels a la
methode
           * dont la signature est citee sera tracee.
           */
          pointcut Postcondition():
             call(public * ControleurClient.*(..))|| execution (public *
FenetreClient.*(..));

          /**
           * Code advice qui  affiche les messages de trace.
           */


          before(): Postcondition() {


            try {
                  String methodeName= thisJoinPoint.getSignature().getName();
                  Object[] args =thisJoinPoint.getArgs();
                  Object caller =thisJoinPoint.getThis();
                  Object callee =thisJoinPoint.getTarget();
                        monFichier = new FileWriter("E:\\Trace2.txt",true);
                        tampon = new BufferedWriter(monFichier);

                        // Ecrit  dans Trace2.txt
                        tampon.write("-> debut methode  "+methodeName);
                        if (methodeName == "Effectuer"){
                              tampon.write(args[3]+" ");
                              Float objectFloat=(Float)args[3];
                              Float x =objectFloat.floatValue();

                        }
                  tampon.write("\n");

                  } catch (IOException exception) {
                  exception.printStackTrace();
                  } finally {
                  try {

                  tampon.flush();
                  tampon.close();
                  monFichier.close();
                  } catch (IOException e1) {
                  e1.printStackTrace();
                  }
                  }
          }
```

```
      after(): Postcondition() {


            try {
                  String methodeName= thisJoinPoint.getSignature().getName();
                  Object[] args =thisJoinPoint.getArgs();
                  monFichier = new FileWriter("E:\\Trace2.txt",true);
                  tampon = new BufferedWriter(monFichier);

                  // Ecrit  dans Trace2.txt

                  tampon.write("-> Apres appel "+methodeName);


                  if (methodeName == "Effectuer"){
                  tampon.write(args[3]+" ");
                  Float objectFloat=(Float)args[3];
                  Float Y =objectFloat.floatValue();
                  Float objectFloat1=(Float)args[1];
                  Float x1=objectFloat1.floatValue();
                if (Y<0){
                  tampon.write("Postcondition non verifier");
                   JOptionPane.showMessageDialog(null, "Postcondition non
verifier", "Opération effectuée", 0);
                  tampon.newLine();
                  tampon.flush();
                  tampon.close();
                  monFichier.close();
                   System.exit(0);
                }
            }
                  } catch (IOException exception) {
                  exception.printStackTrace();
                  } finally {
                  try {
                        tampon.newLine();
                  tampon.flush();
                  tampon.close();
                  monFichier.close();
                  } catch (IOException e1) {
                  e1.printStackTrace();
                  }
                  }
          }

      }
```

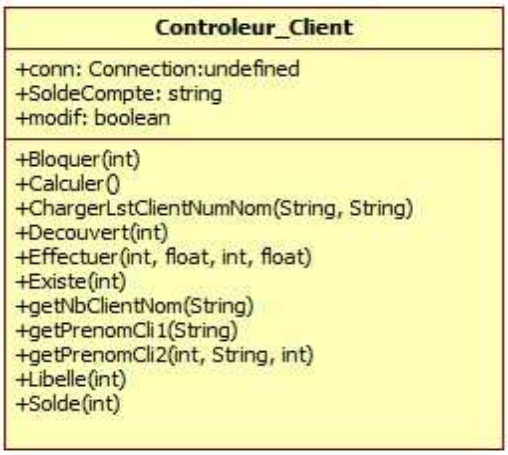
Les advices après le retour sont exécutés. Il vérifie la postcondition et si elle détecte une violation de postcondition . Il exécute :

```
tampon.write("Postcondition non verifier");
JOptionPane.showMessageDialog(null, "Postcondition non verifier", "Opération
effectuée", 0);,
```

sinon retourne normalement.

c) Règle 03 : Transformation d'invariant de classe

Les invariants de classe expriment les propriétés d'une classe qui doit être préservé par tous les routines de la classe Controleur_Client . Invariant de classe est exprimé comme suit:



Context Controleur_Client :
Effectuer(Integer.parseInt(jTxtNum .getText()), montant, y, solde)
inv : solde > 0

Le mot-clé *context* indique le cadre de l'invariant, qui est la classe Controleur_Client dans le cas présent, ***inv*** est indique le type de la contrainte qui est un invariant. L'invariant solde > 0 lui-même est une expression booléenne.

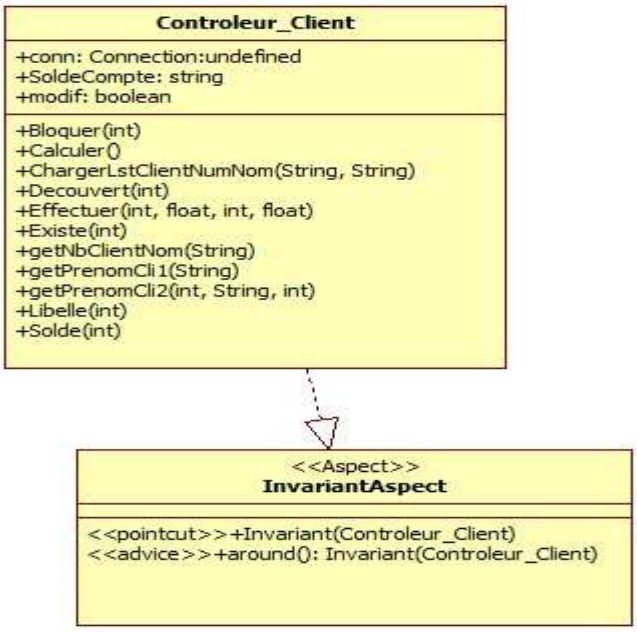


Figure IV.5 Modèle conceptuel représente l'invariant de classe Controleur_Client par AspectJ

Cet invariant est invoquée par *before* et *after advice*, il est traduit à AspectJ dans le code suivant:

```
public aspect InvariantAspect {
       int i=1;
          FileWriter monFichier = null;
            BufferedWriter tampon = null;
          /**
           * Cette coupe est très détaillée puisque seuls les appels a la
methode
           * dont la signature citée sera tracée.
           */
          pointcut Invariant():
             call(public * ControleurClient.*(..))|| execution (public *
FenetreClient.*(..));

          /**
           * Code advice qui  affiche les messages de trace.
           */

          before(): Invariant() {

                  try {
                  String methodeName=
                   thisJoinPoint.getSignature().getName();
                  Object[] args =thisJoinPoint.getArgs();
                  Object caller =thisJoinPoint.getThis();
                  Object callee =thisJoinPoint.getTarget();
                        monFichier = new FileWriter("E:\\Trace3.txt",true);
                        tampon = new BufferedWriter(monFichier);

                        // Ecrit  dans Trace3.txt
                        tampon.write("-> debut methode  "+methodeName);
                        if (methodeName == "Effectuer"){
                              tampon.write(args[3]+" ");
                              Float objectFloat=(Float)args[3];
                              Float x =objectFloat.floatValue();
                            if (x<0){
                              tampon.write("Invariant non vérifier");
                               JOptionPane.showMessageDialog(null,
                          "Invariant non vérifier", "Opération  non effectuée",
                           0);
                              tampon.newLine();
                              tampon.flush();
                              tampon.close();
                              monFichier.close();
                               System.exit(0);
                            }
                         }
                  tampon.write("\n");

                  } catch (IOException exception) {
                  exception.printStackTrace();
                  } finally {
                  try {

                  tampon.flush();
                  tampon.close();
                  monFichier.close();
```

```
            } catch (IOException e1) {
            e1.printStackTrace();
            }
            }
    }

after(): Invariant() {

      try {
      String methodeName= thisJoinPoint.getSignature().getName();
            Object[] args =thisJoinPoint.getArgs();
            monFichier = new FileWriter("E:\\Trace3.txt",true);
            tampon = new BufferedWriter(monFichier);

            tampon.write("-> Apres appel "+methodeName);


            if (methodeName == "Effectuer"){
                  tampon.write(args[3]+" ");
                  Float objectFloat=(Float)args[3];
                  Float x =objectFloat.floatValue();

                if (x < 0){
                  tampon.write("Invariant non verifier");
                   JOptionPane.showMessageDialog(null, "Invariant non
           vérifier", "Opération effectuée", 0);
                  tampon.newLine();
                  tampon.flush();
                  tampon.close();
                  monFichier.close();
                   System.exit(0);
                }
            }

            } catch (IOException exception) {
            exception.printStackTrace();
            } finally {
            try {
                  tampon.newLine();
            tampon.flush();
            tampon.close();
            monFichier.close();
            } catch (IOException e1) {
            e1.printStackTrace();
            }
            }
    }

}
```

6. Diagramme des classes métier UML/AspectJ

Apres l'application de notre stratégie de transformation. Nous avons proposée dans le chapitre III de greffer les aspects dans l'application pour cela nous avons mise à jour le modèle conceptuel (voir Figure IV.02), Dans notre étude de cas nous obtenons le modèle conceptuel (représenté par le diagramme de classes pour AspectJ) illustré dans Figure IV.06.

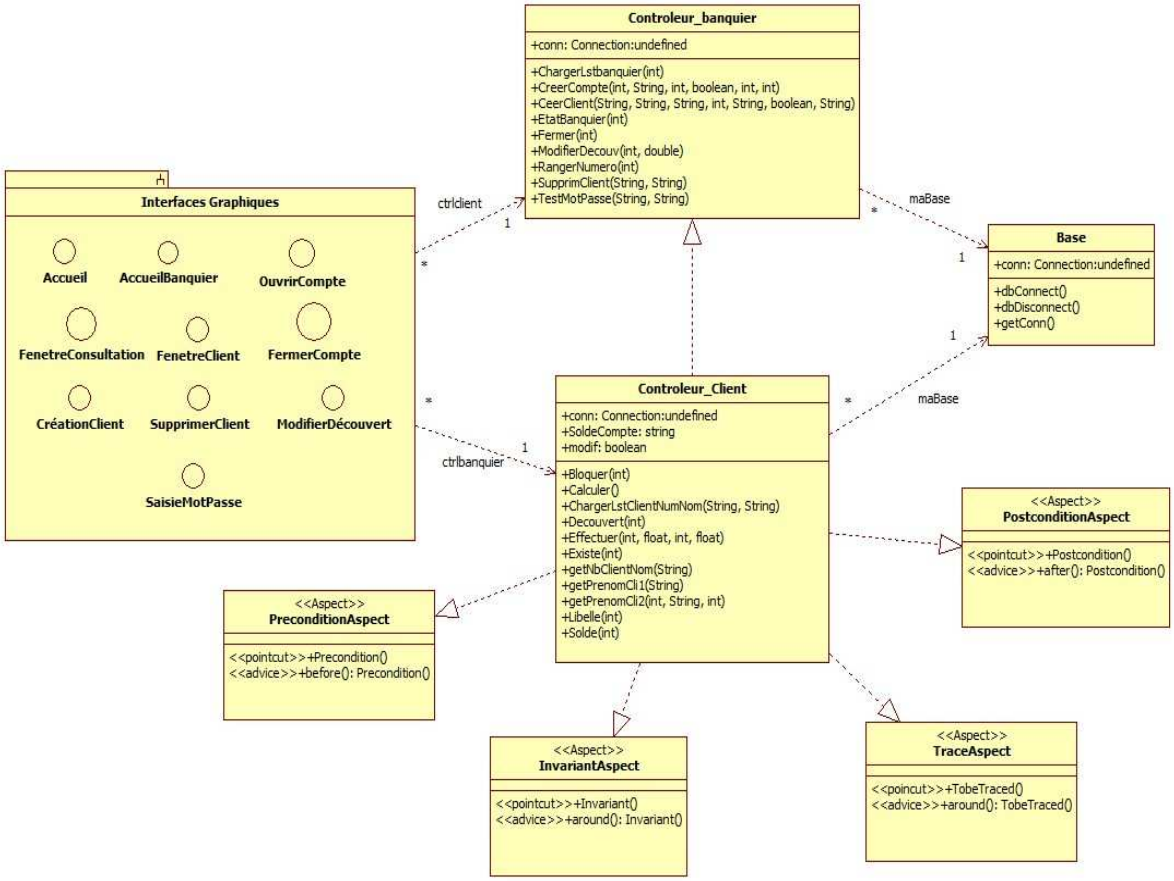


Figure IV.6 *Diagramme des classes UML/AspectJ de gestion bancaire*
Basé sur le modèle étoile

7. Intégration de Design par contrat et test de couverture dans un seul aspect par AspectJ

Nous avons proposé, dans chapitre III, une deuxième approche qui ‘intègre ces techniques en un seul aspect L’intégration des techniques dans un seul aspect nous donnons le modèle conceptuel suivant :

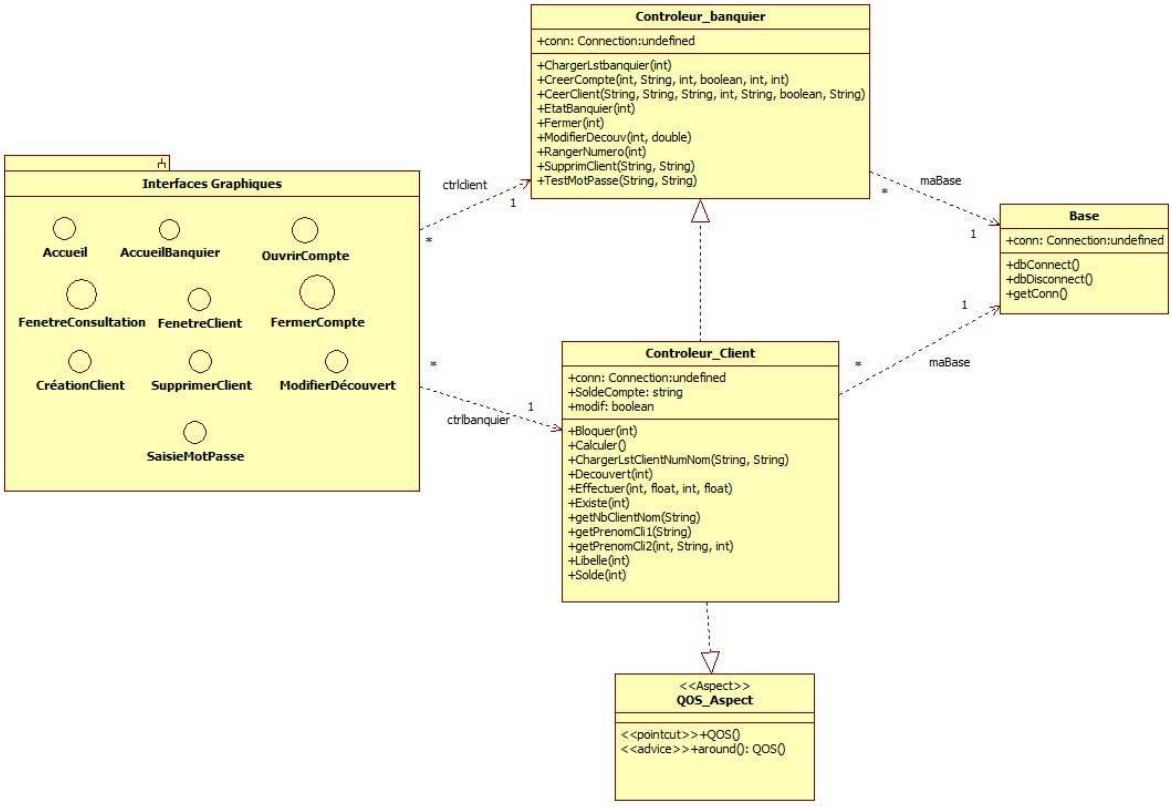


Figure IV.07 *Diagramme des classes UML/AspectJ de gestion bancaire basé sur le modèle arbre*

8. Implémentation des aspects par AspectJ

Nous avons développé notre application, en utilisant le langage Eclipse. Eclipse est un environnement de développement intégré dont le but est de fournir une plate-forme modulaire pour permettre de réaliser des développements informatiques .Eclipse fonctionne uniquement sur les plate formes Windows 98/NT/2000/XP et Linux.

Les interfaces graphiques ont un aspect primaire du fait qu'Eclipse ne propose pas de développement graphique très pointu et efficace.

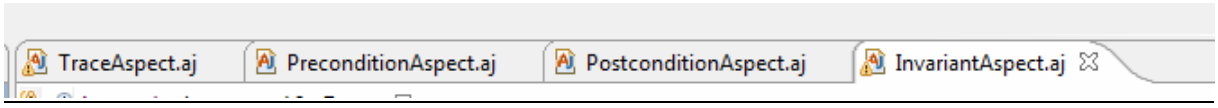


Figure IV.8 *Les aspects de gestion de qualité de service sous eclipse*

La programmation des méthodes de l'aspect peut ce faire avec différent langages comme AspectJ. Pour une bonne vision de l'ensemble de classe de notre application, nous présentant dans ce qui suit les package pour la premier approche (implémenter chaque aspect a part, voir Figure IV.09) et la deuxième approche (un seul aspect de qualité de service, voir Figure IV.10). Les aspects sont marqués par un cercle rouge.

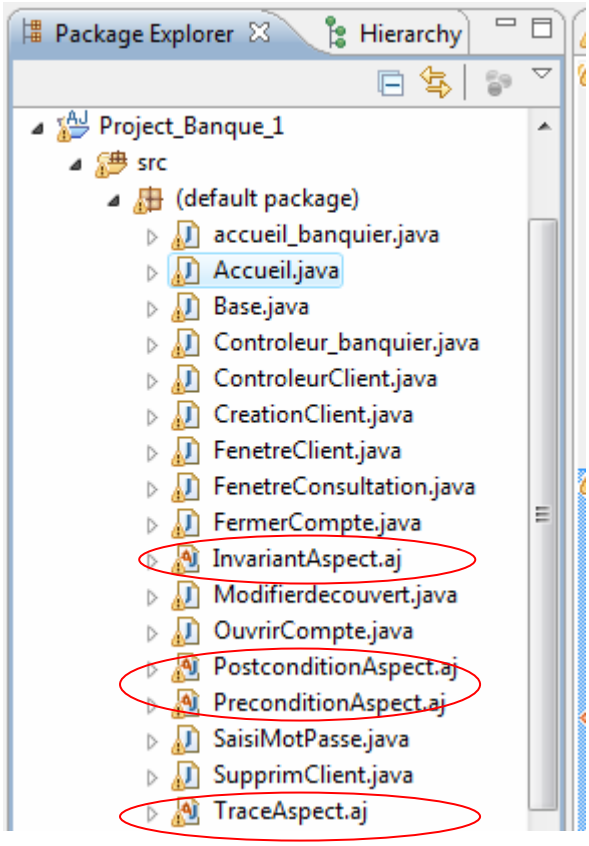


Figure IV.9 *Package des classes et des aspects de Project gestion Bancaire*

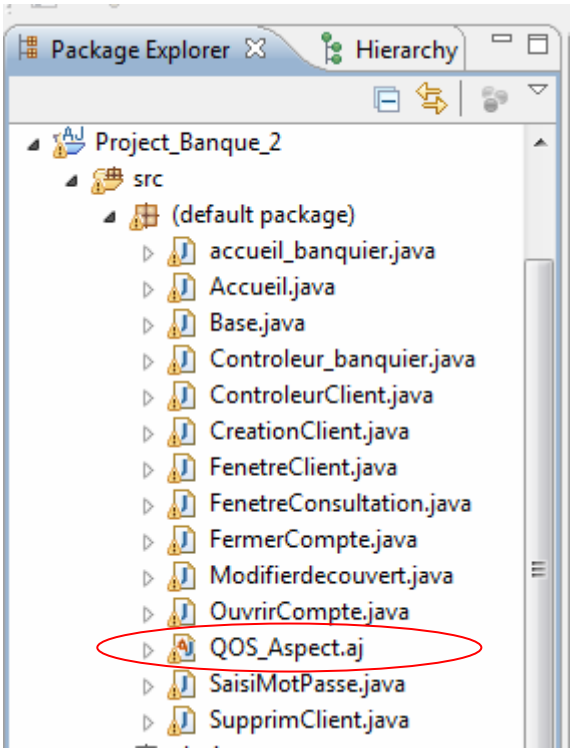


Figure IV.10 *L'aspect de gestion de qualité de service*

Contrairement à toutes les autres classes, la réalisation de la classe Accueil est très importante car elle permet d'orienter les utilisateurs de l'application. La fenêtre d'accueil est présentée dans la figure IV.11.

Figure IV.11 *Fenêtre d'accueil*

Pour facilité l'utilisation de l'environnement AspectJ/Java, nous avons pensé de vous proposer toute les étapes à suivre pour l'exécution de notre application.

- Cliquez deux fois successive sur Accueil.java dans Package des classes et des aspects de Project gestion Bancaire.
- Appuyiez, en suite, sur Run puis Run As.
- Choisissez AspectJ/Java Application (voir figure IV.12) puis valider .

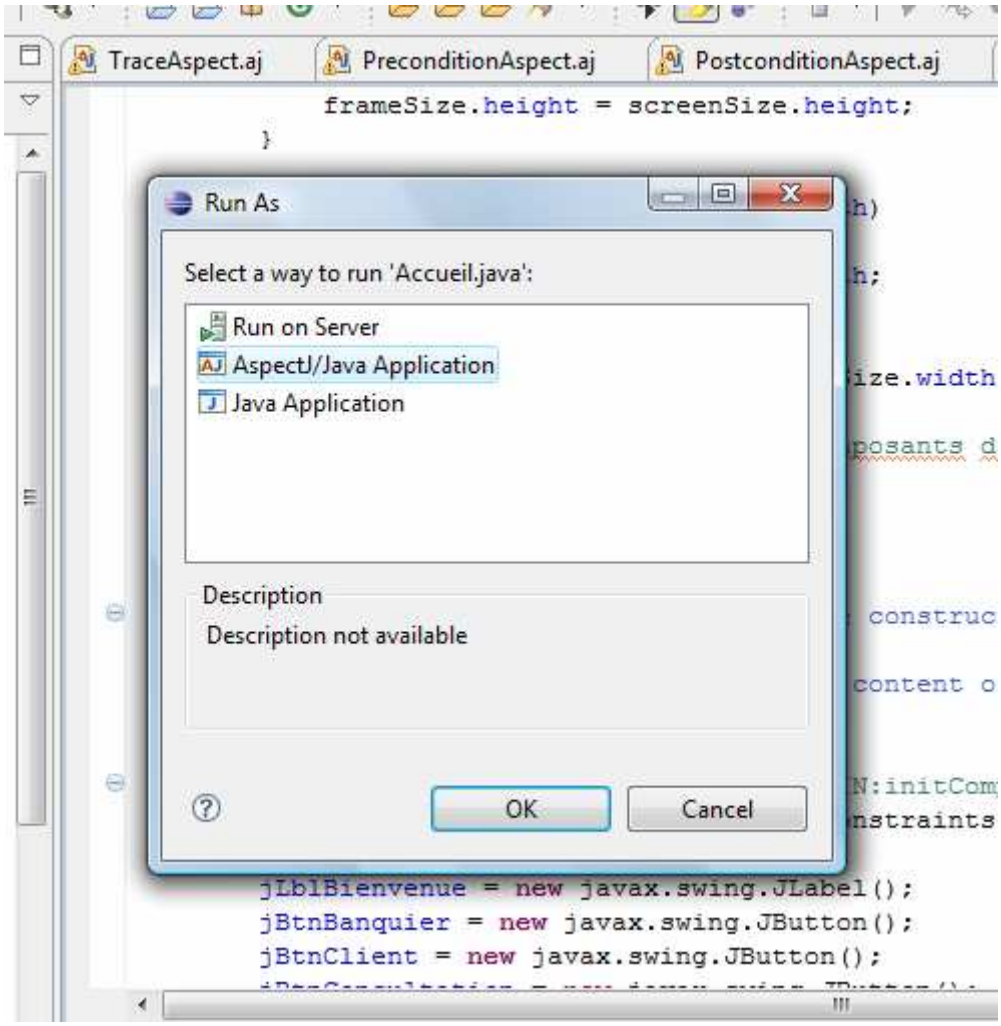


Figure IV.12 *L'étape d'exécution de Project AspectJ/JAVA*

Pour notre application, l'aspect de la précondition intercepte les noms des méthodes et les' écrire sur le fichier de trace (voir Figure IV.14). Si elle intercepte la méthode Effectuer (Integer.parseInt(jTxtNum.getText()), montant, y, solde), elle doit vérifier (montant > 0) avant d'exécuter cette méthode, si elle n'est pas satisfaite, elle conduit d'afficher une boite de dialogue et elle doit arrêter l'application(voir FigureIV.13) .

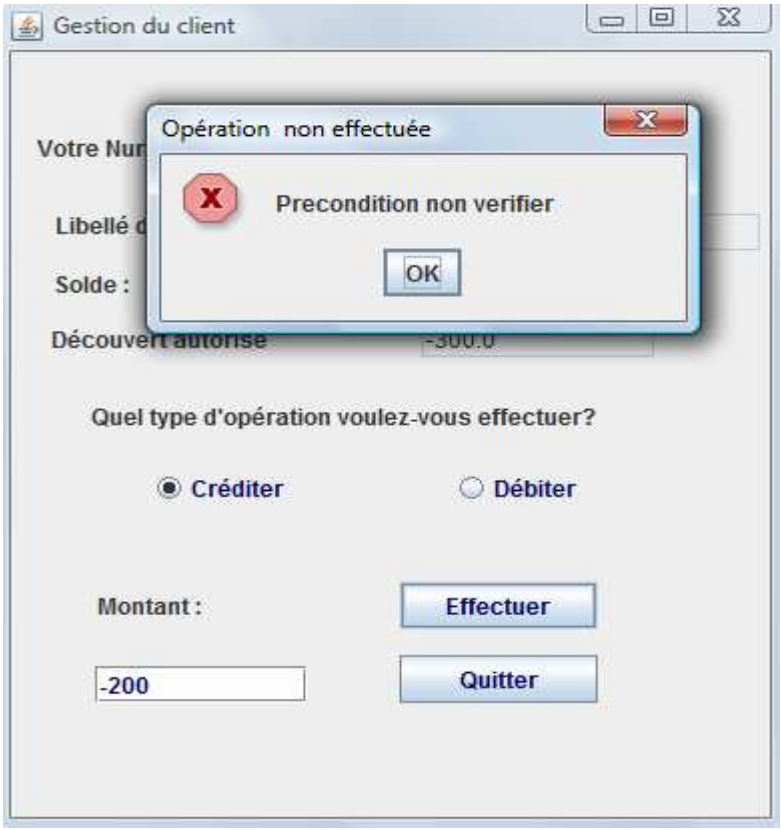


Figure IV. 13 *Exécution d'aspect Precondition non vérifié*

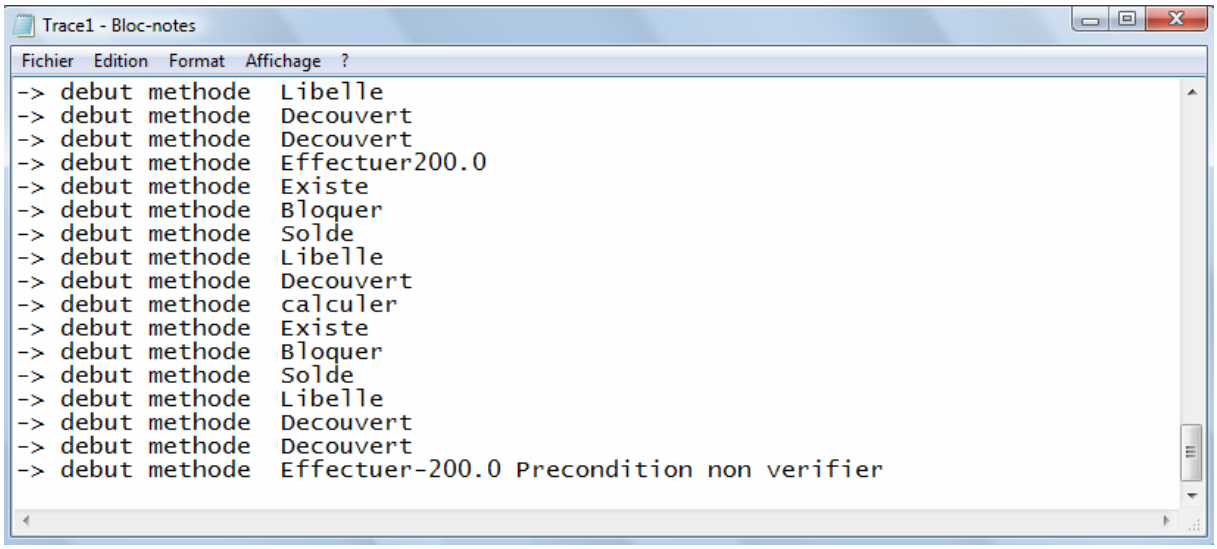


Figure IV.14 *Fichier de trace d'aspect de Precondition*

L'aspect de gestion de trace capture les appels de méthodes dont le code advice serait l'affichage de la trace. La figure IV.15 affiche le fichier de trace de l'exécution de notre application.

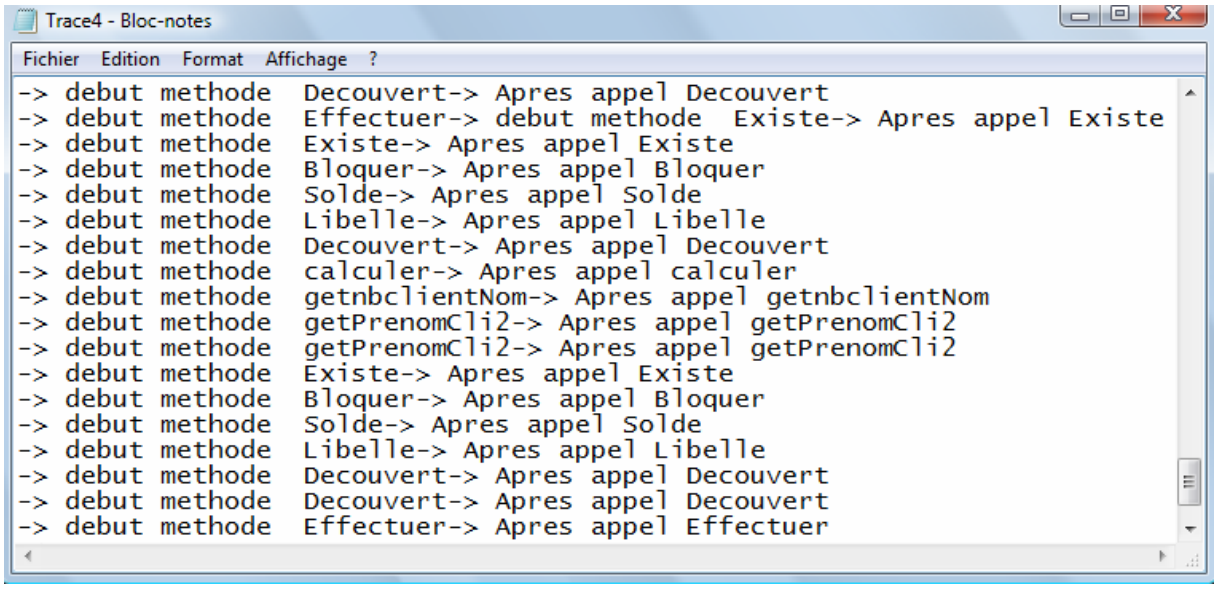


Figure IV.15 *Fichier de trace d'aspect de gestion de trace*

9. Conclusion

Nous avons appliqué dans ce chapitre nos règles de transformation sur une application de gestion bancaire par la programmation orientée aspect.

Nous démontrons que par cette intégration des techniques de gestion de qualité de service nous pouvons augmenter le pourcentage de fiabilité et de performance des logiciels, par un meilleur test, parce qu'il permet de trouver plus des erreurs.

Notre approche a été validée par l'obtention d'une véritable qualité de service. Cette qualité de service parte sur la minimisation de temps de test, la réduction de l'effort fourni ainsi le coût dans le développement des systèmes orientés aspects en comparaison avec l'utilisation de chaque technique à part.

Conclusion générale

Toute application doit être testée afin de garantir à l'utilisateur final une qualité de service qui soit satisfaisante. La programmation orientée aspect est une nouvelle méthodologie qui permet de séparer les préoccupations subsidiaires qui entrecoupent les fonctionnalités principales d'un système. Elle permet la production d 'un code métier « *pur* » découplé du code non-fonctionnel.

C'est une solution aux problèmes de mélange et de dispersion du code des propriétés non fonctionnelles rencontrés avec la programmation par objets, qu'elle consiste à séparer et découpler leurs définitions comme le veut le principe de la *séparation des préoccupations.* Dans le but de permettre une meilleure réutilisation, le paradigme de la *programmation par aspects.*

Nous avons présenté dans cette thèse un état de l'art sur la qualité de service des applications et ses techniques et les langages de programmation par aspects actuellement introduits dans le cadre d'une nouvelle approche de programmation, l'approche Aspect. Nous avons particulièrement présente une étude approfondie de la programmation par aspects (POA) et du langage AspectJ une extension par aspects du langage Java. Les mécanismes et concepts Aspect offerts par ce langage offre une meilleure structuration des programmes, notamment au travers de l'encapsulation et de la séparation des préoccupations transversales au découpage des applications en termes de classes. En palliant les problèmes de dispersion et d'enchevêtrement de code de telles préoccupations, ils présentent ainsi des avantages reconnus quant à l'évolution et à la réutilisation des systèmes logiciels, et permettent par conséquent de s'affranchir de certaines limites de l'approche Objet.

Partant de ces constats, notre travail de recherche a eu plus précisément pour premier objectif de pouvoir localiser le code relatif à la vérification de qualité de service des applications dans des aspects et nous bénéficions de tous les avantages des techniques de gestion de qualité de service, et donc d'améliorer la traçabilités et augmenter en conséquence leur réutilisation, en tirant profit des concepts et mécanismes Aspect.

Nous avons minimisé le temps de test et l'effort ainsi le coût dans le développement des systèmes orientés aspects en comparaison avec l'utilisation de chaque technique appart.

Notre perspective est d'intégrer la troisième technique qui est l'administration et la supervision des applications qui permet de consulter et modifier les paramètres des applications et de surveiller le fonctionnement des applications et qui nécessite un tissage dynamique.

Référence bibliographique

[01] Yishai A. Feldman, Ohad Barzilay, Shmuel Tyszberowicz, "Jose: Aspects for Design by Contract", *the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06),* 2006.

[02] Yu Chin Cheng, Chien-Tsun Chen, and Chin-Yun Hsieh, "ezContract: Using Marker Library and Bytecode Instrumentation to Support Design by Contract in Java", *14th Asia-Pacific Software Engineering Conference*, 2007 IEEE.

[03] Weiqun Zheng and Gary Bundell, "Test by Contract for UML-Based Software Component Testing", *International Symposium on Computer Science and its Applications*, 2008 IEEE

[04] Yves Le Traon, Benoit Baudry, and Jean-Marc Jézéquel, "Design by Contract to Improve Software Vigilance", *IEEE transactions on software engineering,* August 2006.

[05] Yogesh Singh , Anju Saha, "Enhancing Data Flow Testing of Classes through Design by Contract", *Seventh IEEE/ACIS International Conference on Computer and Information Science*, 2008

[06] Guillaume Dufrêne, Simon Morvan,"La programmation Orientée Aspects AspectJ et JAC".

[07] Antoine MAROT," L'animation d'algorithmes en programmation orientée aspect" 2006..

[08] Ferut Térence, Leroy Sébastien," La programmation Orientée Aspects", Juin 2004

[09] Tao Xie, Jianjun Zhao, Darko Marinov, David Notkin, "Detecting Redundant Unit Tests for AspectJ Programs", *17th International Symposium on Software Reliability Engineering (ISSRE'06),*2006

[10] Christoph Gladisch, "Verification-based Test Case Generation for Full Feasible Branch Coverage", Sixth *IEEE International Conference on Software Engineering and Formal Methods,* 2008

[11] Amal Elkharraz, Hafedh Mili, Petko Valtchev," Mining Functional Aspects From Legacy Code", *20th IEEE International Conference on Tools with Artificial Intelligence,* 2008 IEEE

[12] Mark Grechanik, Dewayne E. Perry, and Don Batory, "Using AOP to Monitor and Administer Software for Grid Computing Environments", *the 29th Annual International Computer Software and Applications Conference (COMPSAC'05),* 2005 IEEE

[13] Jun ZHU, Changguo GUO, Quan YIN, Jianlu BO, Quanyuan WU," A Runtime-

Monitoring-Based Dependable Software Construction Method",*The 9th International Conference for Young Computer Scientists,* 2008 IEEE

[14] S.V. Patel , Kamlendu Pandey, "SOA using AOP for Sensor Web Architecture" ,*International Conference on Computer Engineering and Technology*, 2009 IEEE

[15] Lionel, C. Briand Yvan Labiche, Johanne Leduc, "Tracing Distributed Systems Executions Using AspectJ" ,*the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005* IEEE

[16] Mario Luca Bernardi,Giuseppe Antonio Di Lucca ,"testing aspect oriented programs : an Approach based on the coverage of the interaction among advices and methods", *Sixth International Conference on the Quality of Information and Communications Technology* ,2007 IEEE.

[17] Renaud pawlak, jean-philippe Retaillé, lionel seinturier ,Programmation orientée aspect pour Java/J2EE, 2004

[18] David Durand-Christophe Logé , "Spécifications et gestion d'informations de QoS dans les applications réparties par les approches modèles et programmation orientée aspect".

[19] Guadalupe Ortiz, Behzad Bordbar, "Model-Driven Quality of Service for Web Services: an Aspect-Oriented Approach", *International Conference on Web Services,* 2008 IEEE

[20] Richard Schantz, John Zinky, David Karr, David Bakken, James Megquier, Joseph Loyall, "An Object-level Gateway Supporting Integrated-Property Quality of Service".

[21] Allouch Bachir, Trantoul Gilles,'Les nouvelles formes de programmation',juin 2003

[22] Thomas GIL, " la Conception Orientée Aspects 2.1 ", 21 janvier 2006.

[23] H. Rebelo, R. lima, M. Cornélio, S. Soares, and L. Ferreira "Implementing java Modeling Language Contracts with AspectJ", Proceedings of the 23rd Annual ACM Symposium on Applied Computing, Fortaleza, Brazil, 2008, pp. 228-233.

[24] http://ecom.ow2.org/xwiki/bin/view/Main/frjmx

[25] Bounour Nora,Ghoul Said and Atil Fadila ,"A Comparative Classification of Aspect Mining Approaches",2006

[26] Arie van Deursen, Marius Marin ,Leon Moonen, "Aspect Mining and Refactoring"

[27] Assia AIT ALI SLIMANE, Muhammad Usman BHATTI, " Utilisation des services et des aspects pour la réutilisabilité du logiciel d'un automate pour l'analyse de plasma".

[28] Sébastien SAUDRAIS, thèse : "Qualité de Service Temporelle pour Composants Logiciels ",2007.

[29] Karine Macedo De Amorim, thèse : "Modélisation d'aspects qualité de service en UML : application aux composants logiciels", 2004.

[30] Ouafa Hachani, thèse :"Patrons de conception a base d'aspects pour l'ingénierie des systèmes d'information par réutilisation ",2006.

[31] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella, "Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects", IEEE September 2006.

[32] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm et W.G. Griswold ''An Overview of AspectJ". *In Proceedings of ECOOP 2001, European Conference on Object- Oriented Programming*, Budapest, Hungary, LNCS, vol. 2072, Springer, pp. 327-353, 2001.

[33] Thomas DEWAR,Cindy JARY, "La programmation par contrat illustrée par le langage Eiffel" , Octobre 2003

[34] A Framework for QoS Management,José Lino Contreras, Jean Louis,Sourrouille, IEEE2002

[35] http://www.journaldunet.com/developpeur/tutoriel/theo/050210-conception-programmation-par-contrat.shtml

[36] http://www2.lifl.fr/~nebut/ens/svl/contrats.html

[37] http://www.laboiteaprog.com/article-83-4-genie_logiciel_test__verification_et_ validation

[38] Bertrand Meyer, " Conception et programmation orientées objet ",Groupe Eyrolles, 2008, ISBN : 978-2-212-12270-1

[39] http://en.wikipedia.org/wiki/Aspect-oriented_software_development

[39] NGUYEN Manh Tien, Programmation Orientée Aspect, Juillet 2005

[40] Wojciech J. Dzidek, Lionel C. Briand, Yvan Labiche," Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java"

[41] Arnaud Gotlieb, "Test structurel de programmes impératifs",2006

[42] El Ayeb Ines, Mansour Jihene, "JMX ou Java Management Extensions", 2006

[43] A. Hamie," Strategies for Translating UML/OCL Design Models to JAVA/JML Designs", Computer Symposium,Taiwan, 2004.

[44] J. Suzuki and Y. Yamamoto *" Extending UML with Aspects: Aspect Support in the Design Phase".* ECOOP'99 Workshop on Aspect-Oriented Programming, 1999.

[45] Yves Le Traon, *"Le test de logiciels ".* 2002.

[46] BERKANE Mohamed Lamine, *‘‘Un modèle de transformation de design pattern vers des programmes orientes aspects ‘‘*, laboratoire LIRE, Université Mentouri de Constantine 2008.
[47] BOUANAKA Mohamed Ali, “Une méthode de développement de serveurs d’applications basée sur l’approche orientée aspect“, laboratoire LIRE, Université Mentouri de Constantine, 2009.

Glossaire

Ce glossaire regroupe et donne la définition des principaux termes utilisés dans ce mémoire :

POA Programmation Orientée Aspect

DBC Design By Contrat (conception par contrat)

JMX Java Management Extensions

OCL Object Constraint Language

UML Unified Modeling Language

SNMP Simple Network Management Protocol

JVM Java Virtual Machine

HTML Hypertext Markup Language

IACFG Inter-procedural Aspect Control Flow Graph

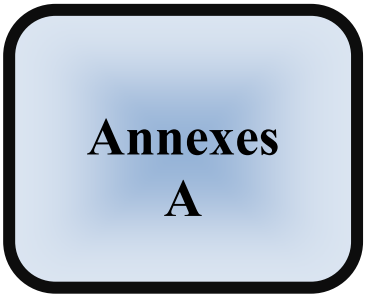
HTTP HyperText Transfer Protocol

JML Java Modeling Language

Annexes

Annexes A

Aperçu sur le logiciel StarUML



Aperçu sur le logiciel StarUML

I. Introduction

StarUML est un logiciel de modélisation UML open source sous une licence modifiée de GNU GPL. L'objectif de ce projet est de se substituer à des solutions commerciales comme IBM Rational Rose ou Borland Together. StarUML gère la plupart des diagrammes spécifiés dans la norme UML 2.0. StarUML est écrit en Delphi ce qui explique en partie pourquoi il n'est plus mis à jour. StarUML est un logiciel de modelage UML qui est entré récemment dans le monde de l'OpenSource.

Le site Internet de StarUML se trouve à http://staruml.sourceforge.net/. Disponible uniquement sous Windows - pour cause d'utilisation d'objets COM -, le téléchargement s'effectue sur http://staruml.sourceforge.net/en/download.php.

Outre l'exécutable, il peut être utile de télécharger l'un ou l'autre module, d'autres sont déjà installés, comme le générateur de code source C++, C#, Java, ... ou le module patterns. Le code est aussi téléchargeable http://staruml.tigris.org/.

Pour finir l'introduction, le logiciel n'est disponible qu'en anglais, et l'aide en anglais ou en coréen.

II. Première utilisation

Après l'installation, StarUML peut être directement lancé. En regardant dans la documentation, on se rend compte que StarUML supporte la version 1.4 d'UML ainsi que la notation de la version 2.0. L'approche MDA est mise en avant par le logiciel de part sa capacité à générer du code à partir d'un modèle UML pour une plateforme .Net par exemple.

Au démarrage, StarUML propose plusieurs nouveaux patrons de projets. Par exemple, les approches Rationale, 4+1 view peuvent être utilisées, chaque approche ouvrant plusieurs diagrammes par défaut. A ce propos, l'approche par défaut comprend un diagramme des cas d'utilisation, d'analyse, de classe, d'implémentation et de déploiement.

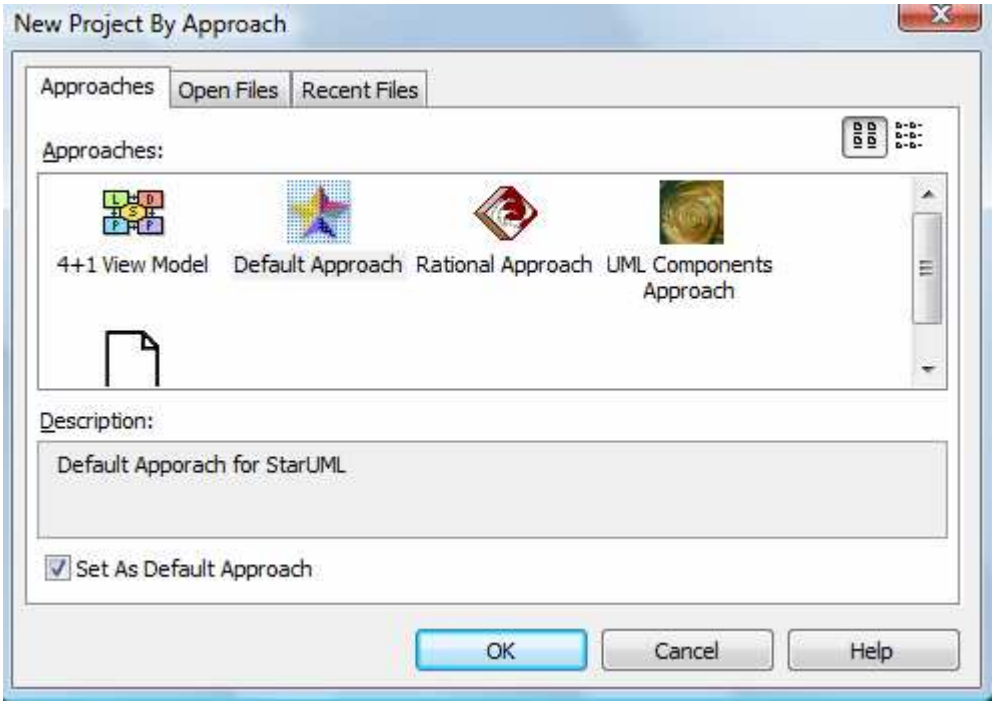


Figure 1 Patron de projets de StarUML

Ceux qui ont l'habitude des Visual Studio ne se sentiront pas perdus non plus, l'interface graphique étant très proche du célèbre IDE de Microsoft.

III. Création d'un projet simple

On va maintenant voir comment StarUML tient le choc dans l'utilisation courante avec le développement d'un projet. On va donc créer un projet avec l'approche de StarUML.

III-A. Use Case

Pour commencer, on va ouvrir la fenêtre des Use Cases. Pour créer un nouveau diagramme, il suffit d'appuyer avec le bouton droit sur le dossier dans lequel créer le nouveau diagramme et sélectionner "Add Diagram". De même, on peut aussi créer un nouveau diagramme dans le menu "Model"->"Add Diagram".

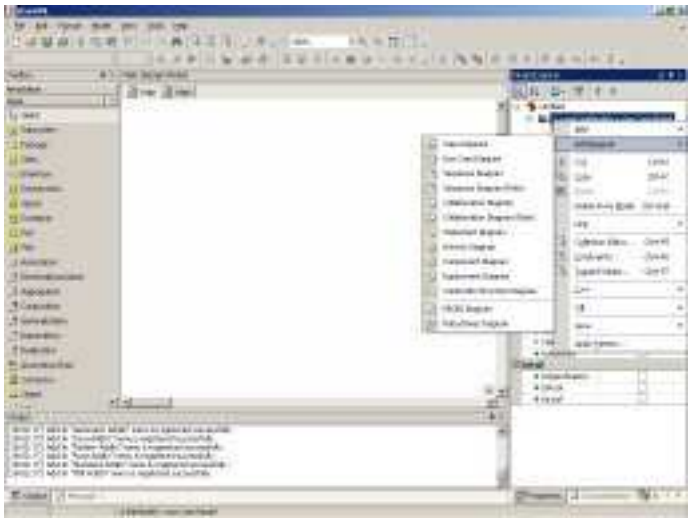


Figure 2 Création d'un nouveau diagramme Use-Case

Enfin, donc on clique sur ce qu'on veut puis on clique sur le diagramme pour les placer. Les noms donnés aux éléments reflètent bien l'origine de StarUML, la Corée du Sud.

En revanche, ce qui est vraiment sympa, c'est la possibilité d'avoir des raccourcis en entrant le nom d'un nouvel élément. Par exemple, pour créer des Use Cases reliés à un acteur, il suffit de modifier le nom de l'acteur en ajoutant "-()" devant le nom du nouveau cas d'utilisation. Cela donne quelque chose de ce genre :

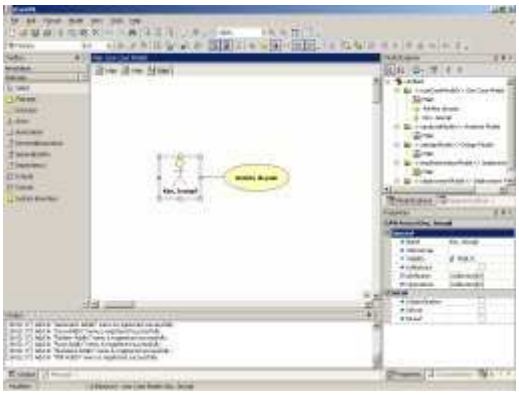


Figure 3 Création d'un Use Case par raccourci

Bien évidemment, il y a d'autres raccourcis, dépendant des diagrammes à créer, mais il faut un peu d'habitude pour s'en servir. On peut aussi créer des associations entre Use Cases et acteurs, on peut créer des généralisations d'Use Cases, d'acteurs, des inclusions, des extensions, ...

III-B. Diagramme de classes

Il existe un type de diagramme que nous n'allons explorer, le diagramme d'analyse qui est une extension du diagramme des classes.

On va donc créer une nouvelle classe **Class1**. Ensuite, pour ajouter des attributs ou opérations, on peut soit les gérer dans l'éditeur de collection - Ctr + F5 ou bouton droit -> éditeur de collection -, soit en double-cliquant sur la classe. Dans ce dernier cas, avec le boutons de gauche, on indique si l'attribut sera **private**, **protected**, ... tandis que les boutons de droite permettent de créer une propriété ou une méthode. Dans l'éditeur de collection, on constate qu'on peut ajouter des paramètres templates, on applaudit, même si c'est un requis d'UML.

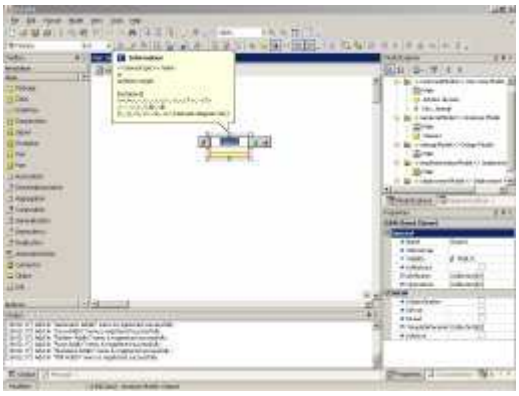


Figure 4 Créer des attributs d'une classe avec le double-clic

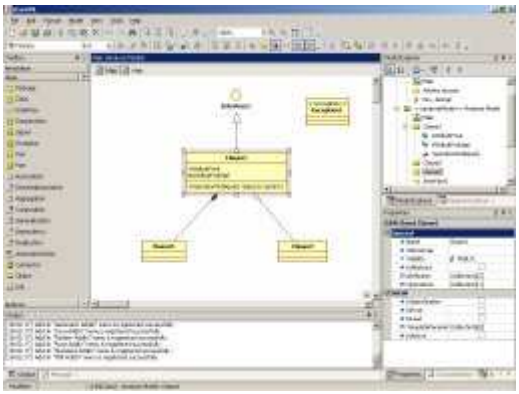


Figure 5 Exemple avec plusieurs classes dans le diagramme

Le but n'étant pas de faire un cours d'UML, on observe simplement la richesse des diagrammes de StarUML. Outre les classiques associations, dépendances, ... on note la simplicité pour ajouter les spécifications d'exceptions, il suffit d'ajouter l'exception, de chercher l'opération dans le Collection Editor et on voit en bas à droite la ligne Raise Signal qui contient la liste des exceptions pouvant être levées.

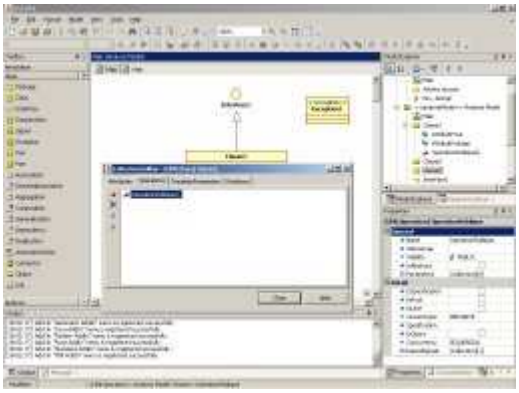


Figure 6 Le champ Raised Signals en bas à droite pour une opération

Les autres diagrammes existent aussi, naturellement, il suffit d'en créer de nouveaux et de faire ce que l'on veut. On peut aussi ajouter des contraintes, celles-ci pourront être écrites dans chaque diagramme dans l'éditeur de contraintes - Ctrl + F6 ou bouton droit -> Constraints -. On note tout de même que l'aide est peu locale sur ces contraintes, à priori le langage utilisé est l'OCL.

Dans chaque diagramme, on peut directement utiliser certains éléments existants dans des diagrammes précédents. Par exemple, les classes des diagrammes de classes sont utilisables dans le diagramme d'implémentation, ou de séquence.

IV. Avantages de StarUML

Le premier avantage est le fait que tous les diagrammes UML 1.x peuvent être générés. Les petits trucs en plus sont importants pour créer des objets dans un diagramme, ajouter rapidement des attributs, ... Le fait que le code source soit disponible est aussi un avantage certain pour l'utilisateur afin d'assurer la continuité du logiciel.
On critique parfois certains diagrammes de séquence dans certains logiciels. Apparemment, celui-ci de StarUML est conforme au standard.

V. Inconvénients de StarUML

Dans les inconvénients, on peut citer le fait qu'il ne soit disponible que sous Windows. De même, l'importation des sources n'est pas parfaite, lorsqu'on ajoute une classe importée dans un diagramme, les connexions avec les autres classes ne sont pas affichées. Enfin, la navigation dans les fenêtres nécessite une habitude qui peut être longue à prendre.
Un inconvénient plus dérangeant est la fâcheuse tendance des flèches à ne pas être droites. Ceci est dû au dimensionnement automatique, donc pour avoir des flèches droites, il faut le retirer, or des flèches non droites ne sont pas visuellement intéressantes, donc le dimensionnement automatique est inutile...

Annexes B

Aperçu sur le logiciel Eclipse

Annexes
B

Aperçu sur le logiciel Eclipse

I. Introduction

Eclipse est un environnement de développement intégré (Integrated Development Environment) dont le but est de fournir une plate-forme modulaire pour permettre de réaliser des développements informatiques.

I.B.M. est à l'origine du développement d'Eclipse. Tout le code d'Eclipse a été donné à la communauté par I.B.M afin de poursuivre son développement. Eclipse utilise énormément le concept de modules nommés "plug-ins" dans son architecture. D'ailleurs, hormis le noyau de la plate-forme nommé "Runtime", tout le reste de la plate-forme est développé sous la forme de plug-ins. Ce concept permet de fournir un mécanisme pour l'extension de la plate-forme et ainsi fournir la possibilité à des tiers de développer des fonctionnalités qui ne sont pas fournies en standard par Eclipse.

Les principaux modules fournis en standard avec Eclipse concernent Java mais des modules sont en cours de développement pour d'autres langages notamment C++, Cobol, mais aussi pour d'autres aspects du développement (base de données, conception avec UML, ...). Ils sont tous développés en Java soit par le projet Eclipse soit par des tiers commerciaux ou en open source. Les modules agissent sur des fichiers qui sont inclus dans l'espace de travail (Workspace). L'espace de travail regroupe les projets qui contiennent une arborescence de fichiers.
Bien que développé en Java, les performances à l'exécution d'Eclipse sont très bonnes car il n'utilise pas Swing pour l'interface homme-machine mais un toolkit particulier nommé SWT associé à la bibliothèque JFace. SWT (Standard Widget Toolkit) est développé en Java par IBM en utilisant au maximum les composants natifs fournis par le système d'exploitation sous jacent. JFace utilise SWT et propose une API pour faciliter le développement d'interfaces graphiques.

Eclipse ne peut donc fonctionner que sur les plate-formes pour lesquelles SWT a été porté. Ainsi, Eclipse 1.0 fonctionne uniquement sur les plate-formes Windows 98/NT/2000/XP et Linux.

SWT et JFace sont utilisés par Eclipse pour développer le plan de travail (Workbench) qui organise la structure de la plate-forme et les intéractions entre les outils et l'utilisateur. Cette structure repose sur trois concepts : la perspective, la vue et l'éditeur. La perspective regoupe des vues et des éditeurs pour offrir une vision particulière des développements. En standard, Eclipse propose huit perspectives.

Les vues permettent de visualiser et de sélectionner des éléments. Les éditeurs permettent de visualiser et de modifier le contenu d'un élément de l'espace de travail.

II. Installation d'Eclipse

Eclipse 1.0 peut être installé sur les plate-formes Windows (98ME et SE / NT / 2000 / XP) et Linux.

Eclipse 2.0 peut être installé sur les plate-formes Windows (98ME et SE / NT / 2000 / XP), Linux (Motif et GTK), Solaris 8, QNX, AIX 5.1, HP-UX 11.

Eclipse 2.1 peut être installé sur toutes les plate-formes sitées précédemment mais aussi sous MAC OS X. Quel que soit la plate-forme, il faut obligatoirement qu'un JDK 1.3 minimum y soit installé. La version 1.4 est fortement recommandée pour améliorer les performances et pouvoir utiliser le remplacement de code lors du débogage. Lors de son premier lancement, Eclipse crée par défaut un répertoire nommé Workspace qui va contenir les projets et les éléments qui les composent.

Le principe pour l'installation de toutes les versions d'Eclipse restent le même et d'une grande simplicité puisqu'il suffit de décompresser le contenu de l'archive d'Eclipse dans un répertoire du système.

- Installation d'Eclipse sous Windows :

Eclipse est téléchargeable sur le site d'Eclipse www.eclipse.org:

la version courante à l'url : http://www.eclipse.org/downloads/

les versions précédentes à l'url : http://archive.eclipse.org/eclipse/downloads/index.php

III. Les points forts d'Eclipse

Eclipse possède de nombreux points forts qui sont à l'origine de son énorme succès dont les principaux sont :

- Une plate-forme ouverte pour le développement d'applications et extensible grâce à un mécanisme de plug-ins.
- Plusieurs versions d'un même plug-in peuvent cohabiter sur une même plate-forme.
- Un support multi langage grâce à des plug-ins dédiés : Cobol, C, PHP, C# , ...
- Support de plusieurs plate-formes d'exécution : Windows, Linux, Mac OS X, ...
- Malgré son écriture en Java, Eclipse est très rapide à l'exécution grâce à l'utilisation de la bibliothèque SWT
- Une ergonomie entièrement configurable qui propose selon les activités à réaliser différentes « perspectives »
- La construction incrémentale des projets Java grâce à son propre compilateur qui permet en plus de compiler le code même avec des erreurs, de générer des messages d'erreurs personnalisés, de sélectionner la cible (java 1.3 ou 1.4
- Une exécution des applications dans une JVM dédiée sélectionnable avec possibilité d'utiliser un débogueur complet (points d'arrêts conditionnels, visualiser et modifier des variables, évaluation d'expression dans le contexte d'exécution, changement du code à chaud avec l'utilisation d'une JVM 1.4, ...)
- Possibilité d'utiliser des outils open source : CVS, Ant, Junit
- La plate-forme est entièrement internationalisée dans une dizaine de langue sous la forme d'un plug-in téléchargeable séparément
- Le gestionnaire de mise à jour permet de télécharger de nouveaux plug-ins ou nouvelles versions d'un plug-in déjà installées à partir de sites web dédiés (Eclipse 2.0).

IV. L’AspectJ pou Eclipse

L’AspectJ Development Tools for Eclipse (AJDT) est une technologie open source Eclipse qui fournit les outils nécessaires pour élaborer et exécuter des applications AspectJ. C’est un bon outil qu’il a un rôle clé à jouer dans la concrétisation des avantages de la programmation orientée aspect, et en particulier en aidant les nouveaux arrivants à comprendre les concepts impliqués.
Le projet AJDT vise à fournir la plate-forme Eclipse pour AspectJ, compatible avec les outils de développement Eclipse Java (JDT), avec des capacités supplémentaires pour la visualisation et la compréhension de la nature transversale de l'aspect des applications orientées. Nous présentons comment installer les outils, comment créer et exécuter des applications AspectJ et la manière de visualiser et de naviguer dans les structures transversales inhérentes à la programmation orientée aspect.

V. Installation et exécution d’AJDT

Pour installer la dernière AJDT pour Eclipse 3.0, procédez comme suit:
1- Lancer Eclipse, puis allez dans Help> Software Updates> Find and Install....
2- Recherche de nouvelles fonctionnalités à installer et à ajouter un nouveau site distant appelé AJDT Mise à jour du site avec http://download.eclipse.org/technology/ajdt/30/update URL.
3- Si vous avez besoin d'un serveur proxy pour accéder à Internet, sélectionnez vos préférences de procuration par Window>Preferences>Install/Update.
4- Développez le nœud AJDT mise à jour du site et sélectionnez AspectJ. Choisir d'installer les outils de développement Eclipse AspectJ, puis lire et accepter la déclaration d'auteur.
5- Cliquez sur Terminer