nne Démocratique et Populaire
Supérieur et de la Recherche Scientifique
Université Larbi Ben Mhidi dOum El-Bouaghi
Faculté de Sciences et Technologie
Département dinformatique

N° d'ordre :

Série :

Ecole Doctorale en Informatique de l'Est

(Pôle Constantine)

Option : Génie Logiciel

MÉMOIRE

Présenté en vue de lobtention du diplôme de

Magister en Informatique

Titre de mémoire :

Une méthode de développement d'architecture logicielle basée sur la notion d'aspect et orientée utilisateur

Présenté par : ELBAZ KHALIL

Dirigé par : Pr. MAHMOUD BOUFAIDA

Président :

Mr Zarour Nacereddine Professeur à lUniversité de Constantine

Rapporteur :

Mr Boufaida Mahmoud Professeur à lUniversité de Constantine

Examinateurs :

Mr Maamri Ramdane Maître de Conférence à lUniversité de Constantine

Mr Mokhati Farid Maître de Conférence à lUniversité dOum El Bouaghi

merciement

Je souhaite, avant toute chose, remercier Dieu pour m'avoir soutenu et permis la réalisation de cette thèse.

Mes premiers remerciements vont en toute logique à Monsieur Mahmoud BOUFAIDA, Professeur à l'Université Mentouri de Constantine qui, pendant ces trois années de thèse a été mon directeur de thèse et mon collaborateur. J'ai pu à ses côtés faire l'expérience d'un chercheur auprès de qui j'ai appris énormément, scientifiquement et humainement parlant. Je ne pense pas être un jour en mesure de lui rendre tout ce qu'il m'a apporté en si peu de temps. Merci Monsieur!

Cette thèse a été jalonnée de rencontres, de discussions et de personnes à qui je voudrais rendre hommage.

J'exprime ma gratitude à Monsieur Necereddine ZAROUR, Professeur à l'Université Mentouri de Constantine, qui m'a fait l'honneur de présider ce jury.

Toute ma reconnaissance également à

Monsieur Ramdane MAAMRI, Maître de Conférence à l'Université Mentouri de Constantine,

Monsieur Farid MOKHATI, Maître de Conférence à l'Université d'Oum El Bouaghi,

pour l'intérêt qu'ils ont bien voulu porter à ce travail en acceptant de l'examiner.

Un grand merci à l'ensemble du personnel du Vice Rectorat D.P.O pour leur accueil chaleureux.

Que seraient ces lignes sans remercier mes plus proches collègues, mes amis: informaticiens et non informaticiens ? Merci à vous pour tous les bons moments, . . .

Je voudrais remercier mon ami, Ismaïl 'Boxic', mon oncle Chouaïb. Leurs remarques bienveillantes et dénuées d'un quelconque caractère informatique m'ont aidé à prendre le recul nécessaire dans les moments difficiles.

Encore quelques lignes, pour exprimer toute la gratitude que j'ai pour mes parents, mon frère et mes sœurs, qui m'ont toujours encouragé et soutenu de manière inconditionnelle dans l'accomplissement de ce travail.

قال الله تعالى:

« إنا فتحنا لك فتحا مبينا (1) ليغفر لك الله ما تقدم من ذنبك و ما تأخر و يتم نعمته عليك و يهديك صراطا مستقيما (2) و ينصرك الله نصرا عزيزا(3) »

الفتح الآيات 1، 2، 3

Table des matières

1 Caractéristiques de la POA

Ce chapitre couvre

- *Introduction*
- *Limites de la POO*
- *Apport de la POA*
- *Présentation du langage AspectJ*
- *Analyse des approches existantes pour la modélisation par aspect*
- *Evaluation des méthodes architecturales*
- *Conclusion*

ractéristiques de la POA

1. Introduction :

Les langages orientés objet, dits de troisième génération, sont devenus depuis les trente dernières années les standards en matière de programmation et de production de logiciels. En voulant modéliser la réalité le plus fidèlement possible, grâce aux concepts de classe et dôobjet, ils ont amené de nouveaux défis pour le domaine du génie logiciel. Les ingénieurs en logiciel ont dû définir de nouvelles méthodes et de nouveaux outils dans le but dôévaluer la qualité des logiciels pour aider aussi bien les gestionnaires que les développeurs. Les travaux de pionniers, tels que Fenton [50] ou encore Chidamber et Kemerer [49], ont révolutionné le domaine en établissant des techniques et des métriques plus adéquates à lôévaluation des programmes orientés objet.

Ces langages de troisième génération ont cependant fait lôobjet de sérieuses critiques. Leur inhabilité à implémenter correctement les préoccupations transversales, telles que la journalisation ou la sécurité, entraîne, entre autres, une mauvaise modularité des programmes orientés objet et les rend difficilement réutilisables. Côest dans le but de palier à ces problèmes et de réaliser une réelle « séparation des préoccupations » [36] quôest né le nouveau paradigme de la programmation orientée aspect (POA) [51].

La programmation orientée aspect a été créée pour capturer les préoccupations transversales à un programme orienté objet. Elle propose dôintroduire le concept *d'aspect* dans ces systèmes. Un aspect possède les mêmes mécanismes quôune classe, ce qui lui permet dôencapsuler une préoccupation aussi bien quôune classe. En tant que paradigme de programmation, la POA vient augmenter la programmation orientée objet et, pour cette raison, nôest pas destinée à être indépendante.

Dans ce chapitre, nous allons donc aborder le point clé de notre solution conceptuelle : la programmation orientée aspect. Nous commencerons par expliquer ses concepts théoriques pour les illustrer, à la prochaine partie, avec le langage AspectJ [48].

voquer les buts de la POA et justifier son utilité. Nous ne nous intéresserons qu'aux apports de la POA en tant qu'extension de la POO. Pour ce faire, nous verrons certaines limites de la POO et nous expliquerons les moyens mis en oeuvre par la POA pour les corriger.

2. Limites de la POO

L'intérêt de la POO pour le développement d'applications complexes est indéniable. Nous allons voir que, dans certains cas, la POO n'est pas techniquement capable de fournir une solution satisfaisante pour aboutir à des programmes clairs et élégants. Ces cas concernent les fonctionnalités transversales et la dispersion de code.

2.1 Fonctionnalités transversales

Comme nous l'avons dit, la POO s'efforce de découper une application en classes cohérentes et indépendantes. Cependant, ces entités peuvent parfois être liées. Pawlak et al [14] nous donnent 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 connaître 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).

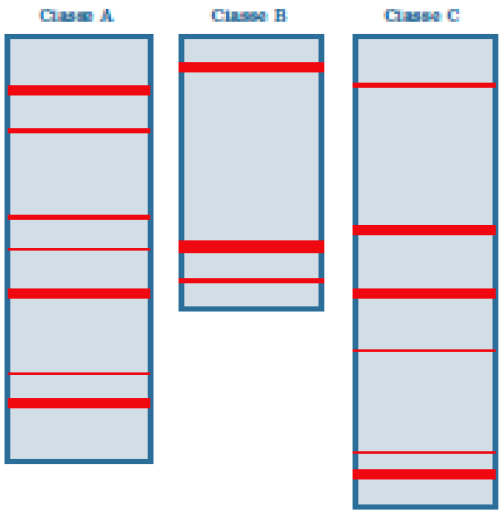


Fig.1.1 – Dispersion du code d'une fonctionnalité

Nous pouvons aussi évoquer une fonctionnalité transversale de multithreading : une commande ne peut être consultée et modifiée par plusieurs threads au même moment. Pour réaliser cette fonctionnalité, la classe commande se retrouve attitrée au rôle de verrouillage de

e devrait sôoccuper uniquement de ce qui concerne les commandes.

Une fonctionnalité de traçage de lôapplication est encore un autre exemple de fonctionnalité transversale. En POO, le code de traçage se retrouvera dans toutes les classes à tracer alors que ce nôest pas une fonctionnalité de leur ressort.

Nous constatons donc que le découpage en classes proposé par la POO ne permet pas toujours lôindépendance de celles-ci.

2.2 Dispersion du code

En POO, 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 quôil est facile de rassembler des services offerts dans une classe, il est impossible de rassembler lôutilisation dôun service.

Ainsi, lôimplémentation dôune méthode est clairement localisée (puisquôelle est dans une classe), alors que les appels vers celle-ci se retrouvent dispersés dans différentes classes (voir fig.1.1). Dès lors, la modification de la méthode est aisée mais pas la modification de sa signature. Une modification de la signature dôune méthode implique la modification de toutes les classes invoquant cette méthode, ce qui rend la maintenance et lôévolution du code difficile.

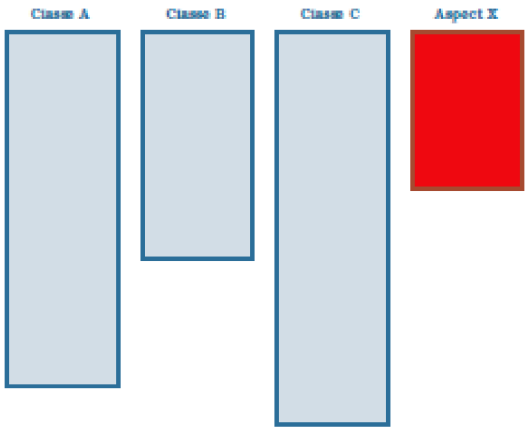


Fig.1.2 – Localisation du code d'une fonctionnalité transversale dans un Aspect

3. Apports de la POA

Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une application. [14]

pporte des solutions aux deux cas que nous avons analysés. Elle 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. Nous expliquerons ces deux concepts plus en détails dans ce chapitre.

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 fig.1.2).

Cette nouvelle notion d'aspect, introduite par la POA, change la façon dont le design d'une application est réalisé.

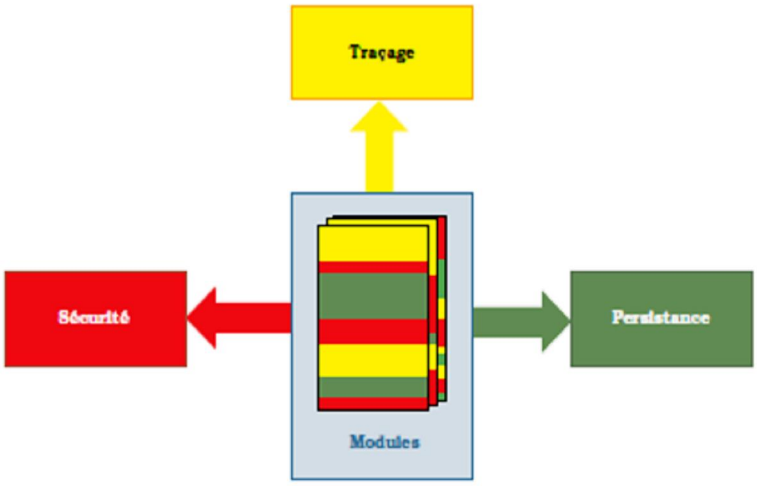


Fig.1.3 – Séparation des facettes d'une application **[18]**

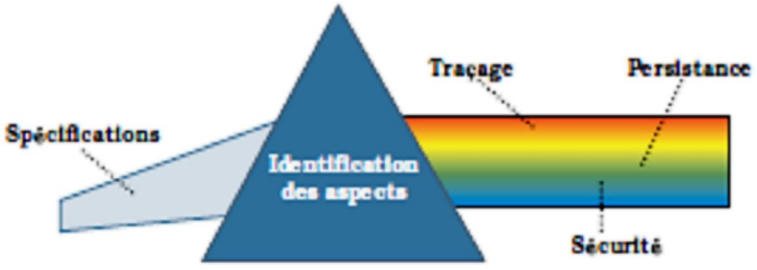


Fig.1.4 – Identification des aspects d'une application **[18]**

Les fonctionnalités transversales sont expulsées du code métier (voir fig.1.3). La première étape du design constitue l'identification des classes en tant que socle de l'application. ***"Il s'agit des données et des traitements qui sont au coeur de la problématique de l'application***

…iers de celle-ci" [14]. Les facettes parallèles du programme ne doivent donc pas se retrouver dans les classes. Elles sont identifiées dans la deuxième partie du design : l'identification des aspects (voir fig.1.4).

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'équipe de développement, pendant que l'aspect persistance sera implémenté par un autre service.

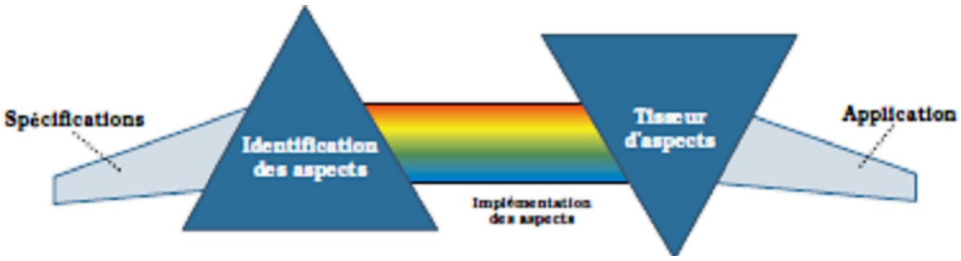


Fig.1.5 – Tissage des aspects **[18]**

3.1 Tissage d'aspects ou Weaving

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 1.5, se nomme le tissage (weaving) et est réalisée par le tisseur d'aspects (aspect Weaver).

Le tissage peut se réaliser soit à la compilation, soit à l'exécution. Lorsqu'il est effectué à la compilation, le tisseur d'aspects se comporte pratiquement comme un compilateur (on le dénomme même parfois compilateur d'aspects). Il prend en entrée les classes et les aspects pour donner en sortie une application tissée. La sortie peut être sous la forme d'exécutable, de byte code ou encore de code source. La dernière solution permettant aux programmeurs d'observer les effets du tissage. L'entrée, quant à elle, peut être sous forme de code source ou de byte code. Une entrée sous forme de byte code permet d'aspectiser une application dont le code source n'est pas connu (une application commerciale par exemple).

Lorsque le tissage est effectué à l'exécution, le tisseur d'aspect se présente sous la forme d'un programme qui permet d'exécuter à la fois l'application et les aspects. Les aspects ne sont donc pas intégrés dans l'application à l'avance et existent encore individuellement à l'exécution. Cette technique est intéressante car il est possible d'ajouter, de modifier ou d'enlever des aspects au *runtime*. Le tissage à l'exécution est aussi appelé tissage dynamique.

pécifier des aspects afin que ceux-ci puissent être

3.2 Points de jonction

Les aspects définissent les endroits du code où ils vont intégrer les fonctionnalités transversales. Cette spécification se réalise à lôaide de coupes qui sont elles-mêmes spécifiées à lôaide 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. [14]

Lôutilisation des points de jonction se fait essentiellement sous forme dôensemble, une coupe définissant tous les points de lôapplication auxquels elle souhaite greffer lôaspect. Ces points de lôapplication peuvent être lôappel dôune méthode, lôexécution dôun constructeur, la lecture dôun attribut.

3.2.1 Types de points de jonction

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

Plusieurs types dôévénements peuvent faire figure de points de jonction.

1. **Méthodes.** Côest autour des méthodes que les aspects se greffent le plus souvent. Ce nôest pas étonnant puisque les méthodes forment lôoutil principal de la POO et structurent lôexécution du programme. Les événements liés aux méthodes qui constituent des points de jonction sont lôappel dôune méthode et lôexécution de celle-ci.
2. **Constructeurs.** Les constructeurs peuvent être considérés comme des méthodes particulières. Il sôagit de méthodes invoquées lorsquôun objet est instancié. Comme pour les méthodes, la POA permet dôintercepter cet événement.
3. **Exceptions.** Les événements de levée et de récupération dôexceptions peuvent aussi constituer des points de jonction. Pawlak et al [14] justifient leurs utilisations pour un cas de gestion centralisée dôune exception par exemple. Le code à exécuter lors de la levée de cette exception ne sera défini quôune seule fois dans un aspect.
4. **Attributs.** La lecture et la modification dôattributs constituent également des points de jonction. On peut penser notamment à une utilisation à des fins de persistance.

3.2.2 Limites des points de jonction

Nous venons de voir certains types dôévénements qui peuvent constituer des points de jonction. Mais la POA peut connaître des limites quant à la granularité correspondante aux points de jonction. En effet, les instructions (if, while, for, switch, ...) sont jugées trop fines

onc pas interceptibles. Cest par exemple le cas pour le langage Aspects que nous verrons dans la prochaine section.

3.3 Coupes

Nous venons de voir la notion de point de jonction sur laquelle se base la notion de coupe que nous allons aborder dans cette section.

Une coupe désigne un ensemble de points de jonction. [14]

Les points de jonction et les coupes sont conceptuellement fort différents : alors quun point de jonction représente un point dans lexécution dun programme, une coupe est un morceau de code défini dans un aspect. Cest à elle quest attribué le rôle de définir la structure transversale dun aspect.

Pour ce faire, une coupe est définie par des mots-clés identifiant des ensembles de points de jonction. Ces ensembles sont unis dans la coupe par des opérations ensemblistes de base (intersection, union et complémentarité).

Les mots-clés utilisés par la coupe désignent chacun un ensemble de points de jonction en spécifiant son type (appel de méthode, lecture dattributs, ...) et une expression qui précise le type (la méthode α, lattribut β, ...). Ces expressions peuvent faire usage de quantificateurs à des fins de généralité (ex : toutes les méthodes dont le nom est μ, toutes les méthodes qui prennent en paramètre une instance de la classe λ, tous les attributs de la classe Φ, ...)

3.3.1 Filtrage

Nous venons de voir quune coupe est définie par un calcul ensembliste sur des ensembles de points de jonction. Nous navons considéré que des ensembles dun certain type, or les outils existants fournissent également des mots-clés identifiant des ensembles ne dépendant pas dun type (ex : tous les points de jonction dans le code dune méthode, tous les points de jonction dune classe, ...). Ceci permet une plus riche expressivité dans la définition de la coupe par un filtrage plus précis.

3.4 Codes Advices

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

Un code advice est un bloc de code définissant le comportement d'un aspect. [14]

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

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***.

Les codes advices peuvent donc être classés en trois types : code advice ***before***, code advice ***after*** et code advice ***around***. Un code advice ***around*** implique l'execution suivante du programme :

1. Exécution du programme
2. Point de jonction rencontré Ÿ Exécution de la première partie du code advice
3. Appel à proceed Ÿ Exécution du point de jonction
4. Exécution de la deuxième partie du code advice
5. Reprise de l'exécution du programme

L'appel à ***proceed***, et donc l'exécution du point de jonction, est facultatif.

Il peut, par exemple, être utile dans un aspect de sécurité de n'exécuter le point de jonction que si certaines conditions sont remplies.

Il est intéressant de noter qu'il existe d'autres types de codes advices que ***before, after*** et ***around.*** AspectJ propose, par exemple, les types ***after returning*** et ***after throwing*** représentants respectivement le retour normal et le retour anormal (avec levée d'exception) du point de jonction.

3.5 Introspection de points de jonction

Nous avons vu qu'un code advice est exécuté dès 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 [48] proposent 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.

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.

3.6 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 a 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.

3.7 Ordonnancement d'aspects

Nous avons vu qu'un aspect greffe des codes advices sur les points de jonction définis par sa coupe. Dès lors, il se peut que plusieurs aspects aient des points de jonction en communs dans leur coupe. Or dans certains cas, il est nécessaire qu'un code advice soit exécuté avant un autre (si il y a des dépendances entre aspects).

Les outils de la POA fournissent à cet effet des techniques permettant de spécifier l'ordre dans lequel doivent être tissés les aspects. Si le programmeur ne spécifie pas d'ordre, aucune garantie n'est donnée en général quant à l'ordre résultant après le tissage. Cependant, certains outils tels qu'AspectJ proposent des règles implicites d'ordonnancement d'aspects.

Les codes advices peuvent recourir à l'introspection de point de jonction pour identifier le contexte qui a provoqué leur exécution.

Nous avons ensuite considéré le mécanisme d'introduction qui permet d'étendre les classes afin d'y intégrer des attributs, des méthodes, etc... dans le but de faciliter l'écriture des fonctionnalités transversales.

Pour terminer, nous avons vu que l'ordre dans lequel les aspects sont tissés a son importance et ne doit pas être négligé.

lspectJ

Nous venons de présenter dans la section précédente, les principes de la POA dôun point de vue conceptuel et général. Nous allons maintenant nous intéresser à sa réalisation en pratique avec le langage AspectJ [48]. Nous reprendrons point par point les concepts présentés précédemment et nous les illustrerons avec ce langage.

4.1 Aspects

La définition dôun aspect avec AspectJ est très similaire à la définition dôune classe ou dôune interface en Java [18]. Ainsi, tout comme les classes et les interfaces, les aspects portent un nom et peuvent être définis au sein de packages.

AspectJ introduit le mot-clé aspect qui permet de spécifier que lôentité logicielle définie est un aspect, à lôinstar du mot-clé class qui spécifie la définition dôune classe. Lôexemple suivant illustre la définition dôun aspect :

```
package edu.ulb;
public aspect PremierAspect {
/* code de lôaspect */
}
```

La partie /* code de lôaspect */ comprend aussi bien la définition des coupes et des codes advices que lôintroduction de nouveaux éléments, la spécification dôun ordonnancement dôaspects et la déclaration de variables ou méthodes propres à lôaspect. Ces concepts seront explicités au cours de ce chapitre.

Un aspect est défini dans un fichier qui, à lôinstar des classes et des interfaces, porte le même nom que le nom de lôaspect. Ce fichier peut très bien porter lôextension **.java**. Il est préférable que seuls les éléments constituant le code métier de lôapplication portent cette extension. Il est fortement conseillé dôutiliser les extensions .aj ou .ajava pour des aspects.

4.2 Tissage d'aspects

AspectJ est un langage orienté aspect pour Java qui est tissé à la compilation [18]. Les aspects sont tissés directement dans le bytecode de lôapplication. On obtient donc un ensemble de fichiers .class contenant lôapplication aspectisée et compatibles avec la machine virtuelle Java.

Les premières versions dôAspectJ proposaient un tissage de type Java vers Java qui, comme nous lôavons vu au point 3.1, permet dôobserver les résultats du tissage directement dans la

énient dôêtre lourde (on dédouble lôanalyse syntaxique p.e.) et ne permet pas dôaspectiser des applications déjà compilées.

4.3 Points de jonction

Nous avons vu quôun ensemble de points de jonction est identifié dans une coupe à lôaide dôun mot-clé. Une expression est également fournie pour filtrer lôensemble obtenu. Dans cette section, nous verrons les mots-clés disponibles dans AspectJ pour chacun des types de point de jonction et nous détaillerons ensuite la manière de définir les expressions passées en paramètre à ceux-ci.

4.3.1 Types de point de jonction

Dans le chapitre précédent, nous avons considéré les types de points de jonction suivants : méthode, constructeur, exception et attribut. Nous allons voir les mots-clés proposés par AspectJ pour les identifier et nous verrons également quôAspectJ introduit de nouveaux types de points de jonction.

a) Méthodes :

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

- 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 lôexpression methexpr.

La différence fondamentale entre ces deux ensembles de points de jonction est le contexte dans lequel se trouve lôapplication lors de lôexécution du code advice associé. Un point de jonction donné par call (methexpr) correspond à lôappel dôune méthode compatible à methexpr. Il se trouve donc dans le code de la méthode appelante, alors quôun point de jonction donné par execution (methexpr) se trouve dans la méthode appelée.

Afin de mieux illustrer notre propos, imaginons que lôon ait une méthode qui soit associée à un point de jonction call et `a un point de jonction execution. Lôordre dôexécution 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*.

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*.

Remarquons quôun code advice de type *before* (respectivement *after*) nôa pas de deuxième (respectivement première) partie.

b) Constructeurs

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

- Exécution de constructeur : initialization (constrexpr)

initialization (constrexpr) identifie toutes les exécutions dôun constructeur dont le profil vérifie constrexpr.

- Exécution de constructeur hérité : preinitialization (constrexpr)

Ce mot-clé identifie toutes les exécutions dôun constructeur hérité dont le profil correspond à lôexpression constrexpr.

Lôexécution dôun constructeur hérité se fait en Java avec lôaide du mot clé super (..) où représente les paramètres fournis. Java impose que ce genre dôappel soit la première instruction dôun constructeur. Alors que initialization identifie lôexécution de constructeurs, preinitialization identifie lôexécution de constructeurs appelés via cette instruction.

Il est intéressant de remarquer que lôon peut intercepter les appels de constructeur avec le mot-clé call que nous avons explicité précédemment. Il suffit de fournir une methexpr identifiant les méthodes de nom *new*. Dans sa version actuelle, AspectJ ne permet pas la définition dôun code advice de type around sur ces points de jonction.

c) Exceptions

- Récupération d'exception : handler (exceptexpr)

Ce mot-clé identifie toutes les récupérations dôexception dont le profil vérifie exceptexpr. Il sôagit donc, en Java, de lôexécution des blocs catch.

AspectJ ne permet pour le moment que lôutilisation de codes advices de type *before* sur les points de jonction de récupération dôexception.

d) Attributs

et permettent dôintercepter respectivement les lectures et les écritures des attributs.

- Lecture dôattribut : get (attrexpr)

get(attrexpr) identifie tous les points de jonction représentant la lecture dôun attribut dont le profil vérifie attrexpr.

- Modification dôattribut : set (attrexpr)

Ce mot-clé identifie toutes les modifications dôun attribut dont le profil est compatible à attrexpr.

Ces mots-clés sont intéressants pour des aspects qui souhaitent intercepter la modification de lôétat dôun objet.

e) Autres

AspectJ introduit deux nouveaux types de point de jonction :

- Initialisation de classe : staticinitialization (classexpr)

Ce mot-clé identifie lôexécution des blocs statiques dôinitialisation dôune classe correspondant à lôexpression classexpr. Ces blocs sont exécutés automatiquement par la machine virtuelle lors du chargement de la classe.

- Exécution de code advice : adviceexecution ()

adviceexecution () identifie toutes les exécutions dôun code advice. Il ne requiert pas dôexpression en paramètre. Il identifie simplement toutes les exécutions de code advice. Il doit donc être utilisé avec précaution car il peut vite mener `a des boucles infinies dans lôexécution du programme. En effet, le code advice exécuté lors de lôexécution dôun code advice appartient, lui aussi, à lôensemble défini par adviceexecution.

Ce mot-clé doit donc être utilisé en parallèle avec des mots-clés de filtrage, que nous verrons dans la partie consacrée aux coupes.

4.3.2 Définition des profils

A lôexception de adviceexecution, tous les mots-clés que nous avons vus requièrent un paramètre. Ce paramètre est une expression qui permet, en spécifiant un profil, de filtrer lôensemble de points de jonction donné par le mot-clé. Par exemple, parmi tous les points de jonction de type appel de méthode, nous ne souhaitons garder que les appels vers une méthode dont le nom est α.

Lôexpression précisant le profil des éléments qui nous intéresse peut faire usage de quantificateurs (appelés wildcards) afin dôintroduire de la généricité dans les profils sélectionnés. Ces wildcards sont *, et +.

Le symbole * peut être utilisé pour remplacer des noms de classe, de méthode et dôattribut. Il peut soit remplacer lôentièreté du nom, soit une partie de celui-ci.

Exemples :

– public void edu.ulb.Class.*(String)

représente toutes les méthodes publiques de la classe Class (dans le package edu.ulb) prenant un String en paramètre et ne retournant rien.

– public * edu.ulb.Class.*add*(String)

représente toutes les méthodes publiques de la classe Class (dans package edu.ulb) prenant un String en paramètre et dont le nom contient la chaîne add.

– private edu.*.Class.*

représente tous les attributs privés des classes Class qui appartiennent a un sous-package de premier niveau du package edu.

Mais le symbole * peut également être utilisé pour remplacer les propriétés dôune méthode (public, private, protected, static, final, ...).

Exemples:

– * * edu.ulb.Class.*(String)

représente toutes les méthodes de la classe Class (dans le package edu.ulb) prenant un String en paramètre.

– * edu.ulb.Class.*(String)

est identique à lôexemple précédent. Le double * peut être simplifié par un symbole * unique.

b) Le wildcard ..

Le symbole**..** permet de prendre en compte le polymorphisme des méthodes. En effet, les paramètres dôune méthode peuvent être omis à lôaide de ce symbole.

Exemples :

ï public void edu.ulb.Class.*(..)

représente toutes les méthodes publiques de la classe Class (dans le package edu.ulb) ne retournant rien.

ï public edu.ulb.Class.new(..)

représente tous les constructeurs de la classe Class (dans le package edu.ulb)

Le wildcard .. peut également être utilisé pour introduire de la généricité dans la hiérarchie des packages.

ï edu..Class.*(..)

représente toutes les méthodes de toutes les classes Class dans nôimport quel package de la hiérarchie edu.

ï * edu..*.*(..)

représente toutes les méthodes de toutes les classes de nôimporte quel package de la hiérarchie edu.

c) Le wildcard +

Le symbole + est utilisé en tant quôopérateur de sous-typage. En effet, mis en suffixe dôun nom de classe, il permet dôidentifier lôensemble contenant cette classe et toutes ses sous-classes. Sôil est mis à la suite dôun nom dôinterface, il identifie toutes les classes implémentant lôinterface.

Exemples :

ï public void edu.ulb.Class+.*(..)

représente toutes les méthodes publiques de la classe Class (dans le package edu.ulb) et de toutes ses sous-classes, ne retournant rien.

ï * java.awt.Shape+.*(..)

représente toutes les méthodes des classes implémentant lôinterface java.awt.Shape.

4.4 Coupes

Au point précédent, nous avons vu les mots-clés et les expressions qui permettent dôidentifier des ensembles de points de jonction dans lôapplication.

Une coupe est un ensemble de points de jonction résultant dôun calcul ensembliste sur ces ensembles. Les opérations ensemblistes de base disponibles sont lôunion, lôintersection et la complémentarité. Elles sont respectivement représentées dans AspectJ par les symboles | |, && et ! [32].

Exemples :

ï call (* edu.ulb.Class.*(..)) && !call (* edu.ulb.Class.*foo*(..))

représente lôensemble des appels de méthode, dont le nom ne contient pas foo, appartenant à la classe Class (dans le package edu.ulb).

ï call (* edu.ulb.Class1.*(..)) || call (* edu.ulb.Class2.*(..))

représente lôensemble des appels de méthodes appartenant `a lôensemble des méthodes des classes Class1 et Class2 (dans le package edu.ulb).

être réutilisée. Dans ce cas, elle doit être précédée du mot clé pointcut. Mais elle peut également être directement définie dans un code advice, nous donnerons un exemple de ce cas lorsque nous verrons les codes advices au point 4.5.

Exemple :

ï pointcut nomCoupe () : call (* edu.ulb.Class.*(..));

La coupe spécifiée porte le nom nomCoupe(). Les parenthèses sont obligatoires car nous verrons qu'il est possible de paramétrer une coupe.

4.4.1 Filtrage

AspectJ propose des mots-clés identifiant des ensembles de points de jonction indépendants d'un type quelconque. Ils sont, la plupart du temps, utilisés à des fins de filtrage grâce à l'opération ensembliste d'intersection.

Ils permettent ainsi d'affiner l'ensemble obtenu. Ces mots-clés sont détaillés dans le tableau suivant :

withincode (methexpr)	Identifie l'ensemble des points de jonction se trouvant dans une méthode dont le profil vérifie methexpr.
within (typeexpr)	Identifie l'ensemble des points de jonction se trouvant dans une classe ou une interface dont le profil vérifie typeexpr.
This (typeexpr)	Identifie l'ensemble des points de jonction dont le profil de l'objet source vérifie typeexpr. Exemple : l'objet réalisant l'appel d'une méthode dans un point de jonction de type call.
target (typeexpr)	Identifie l'ensemble des points de jonction dont le profil de l'objet destination vérifie typeexpr. Exemple : l'objet sur lequel est appelée une méthode dans un point de jonction de type call.

Il existe encore deux mots-clés qui méritent plus d'attention. Il s'agit des mots-clés *cflow* et *cflowbelow.* Ces mots-clés permettent d'introduire des filtrages basés sur le flot de contrôle.

Les opérateurs que nous avons vus jusqu'ici (withincode, within, this et target) peuvent être qualifiés d'opérateurs statiques. En effet, ils ne dépendent pas de la façon dont s'exécute le programme.

cflow et *cflowbelow* prennent en compte la dynamique du programme. Nous allons, dans un premier temps, nous intéresser uniquement à *cflow* car *cflowbelow* a un comportement presque identique au précédent.

L'opérateur *cflow* prend une coupe en paramètre. Il identifie tous les points de jonction situés entre le moment où l'application passe par un des points de jonction de la coupe et le moment

onction. Lôensemble de points de jonction fourni par *cflow* contient également les points de jonction de la coupe.

Illustrons notre propos par un exemple, imaginons que la coupe c, fournie en paramètre à *cflow*, soit une coupe identifiant tous les appels à une méthode *foo*. *cflow(c)* identifie dès lors, outre les appels à *foo*, tous les points de jonction rencontrés pendant lôexécution de *foo*. La méthode *foo* pouvant faire appel à dôautres méthodes, leur contenu sera aussi considéré.

La seule différence entre *cflow* et *cflowbelow* est le fait que le deuxième cité ne considère pas les points de jonction fournis dans la coupe.

Pour finir, AspectJ propose une dernière technique de filtrage basée sur des expressions booléennes. Il est en effet permis dôutiliser lôopérateur if dans la définition dôune coupe.

Par exemple, if(thisJoinPoint.getArgs().length!=0) identifie tous les points de jonction dont le nombre dôarguments nôest pas nul. Cet exemple utilise les fonctionnalités dôintrospection de point de jonction que nous verrons au point 4.6.

4.4.2 Paramétrage des coupes

Nous avons vu que, lorsquôune coupe est nommée, elle est suivie par des parenthèses. Ceci est dû au fait quôil est possible de paramétrer les coupes.

Les paramètres sont, dans ce cas, des informations qui sont transmises de la coupe vers les codes advices qui lôutilisent.

Les paramètres se définissent à la manière des paramètres dôune méthode (ils sont donc typés) et ils peuvent transmettre trois types dôinformations : lôobjet source, lôobjet cible et les arguments des points de jonction de la coupe. Lôobjet source et lôobjet cible sont transmis à lôaide de mots-clés que nous avons déjà vus : this et target. Cependant, ils sont légèrement différents car ils ne prennent plus une expression désignant une classe en paramètre, mais un nom de variable.

Les arguments des points de jonction désignés par la coupe sont, quant à eux, transmis à lôaide du mot-clé args.

Exemple :

ï pointcut example(Rectangle src, Point cib, int x, int y): call(* *..*(..)) && this(src) && target(cib) && args(x,y);

Représente tous les appels de méthode où lôobjet appelant est un Rectangle,

ramètres de la méthode son deux entiers. Les variables
s les codes advices liés à cette coupe.

Il est important de noter que ces informations peuvent également être connues par un code advice grâce à lôintrospection de point de jonction que nous verrons au point 4.6.

4.5 Codes Advices

Dans la section précédente, nous avons vu que la POA propose trois types de codes advices : before, after et around. AspectJ introduit deux nouveaux types : *after returning* et *after throwing.* Ces deux types supplémentaires représentent respectivement le retour normal et anormal (avec levée dôexception) dôun point de jonction.

Nous allons détailler chacun des types de code advice proposés par AspectJ dans cette section.

4.5.1 Le type before

Un code advice est associé à une coupe. Cette coupe peut être utilisée par plusieurs codes advices si celle-ci est nommée. Lôexemple suivant donne un exemple de code advice de type *before* dont la coupe est nommée :

```
/* 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 òAvant un appel de méthodeò avant chaque appel de méthode de lôapplication. Lôexemple suivant défini le même comportement mais sans faire usage dôune coupe nommée :

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

4.5.2 Le type after

Alors quôun code advice de type *before* sôexécute avant les points de jonction de sa coupe, un code advice de type *after* sôexécute après ceux-ci. Vu quôil sôagit de lôunique différence entre

ûils aient une syntaxe pratiquement identique : il suffit dûutiliser le mot-clé *after* à la place de *before*.

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

4.5.3 Le type around

Les codes advices de type *around* exécutent du code avant et après les points de jonction. Les deux parties sont séparées dans le code advice par le mot-clé proceed. Ce mot-clé a pour but dûexécuter le point de jonction courant. Il se peut très bien que proceed ne soit pas appelé par le code advice, ceci impliquant que le point de jonction ne sera pas exécuté.

Contrairement aux types *before* et *after*, le type *around* possède un type de retour qui correspond au type de retour des points de jonction. Par exemple, le type de retour dûun point de jonction appel de méthode correspond à celui de la méthode, le type de retour dûun point de jonction écriture dûattribut est void, ... Il se peut également que les points de jonction de la coupe ne soient pas tous du même type. Il faut alors renvoyer un type qui soit surtype de lûensemble des types des points de jonction.

Illustrons les concepts présentés par quelques exemples. Lûexemple suivant montre que lûexécution de *proceed()* génère une valeur, la valeur générée par le point de jonction.

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

Le prochain exemple illustre lûutilisation dûun code advice de type *around* dans le cas dûune écriture dûattribut. Cette écriture étant une instruction qui ne renvoie rien, le type de retour du code advice est void.

```
void around(): set(double Class.ratio) {
System.out.println("avant modification du champ ratio");
proceed();
System.out.println("après modification du champ ratio");}
```

dôutilisation du code advice *around* à des fins de sécurité. Le point de jonction nôest exécuté que si lôaccès est autorisé. La méthode *method* retournant un double, notre code advice en renvoie également un.

```
double around(): call(double Class.method(..)) {
 if(access_granted) {
 return proceed();
 }
 else {
 return 0;
 }
}
```

Chacune des parties séparées par *proceed* peut être vide, on obtient dès lors un comportement semblable à un code advice de type *before* ou *after* selon la partie qui nôexiste pas. Mais en réalité, il est possible dôavoir plus de deux parties dans un code advice de type *around* car il est permis dôappeler plus dôune fois proceed. Ce genre de cas est plutôt rare.

Lorsque la coupe dôun code advice de type *around* est paramétrée, il est nécessaire de fournir tous les paramètres de celle-ci à la méthode *proceed()*.

4.5.4 Le type after returning

Les codes advices de type *after returning* sôexécutent à chaque terminaison normale dôun des points de jonction de la coupe. Il est possible de récupérer la valeur retournée (si elle existe) en paramétrant le code advice.

Côest ce quôillustre le code suivant :

```
after() returning (double d): call(double Class.method(..)) {
System.out.println("method(..) a retourné : " + d);
}
```

4.5.5 Le type after throwing

Les codes advices de type *after throwing* sôexécutent à chaque terminaison anormale dôun des points de jonction de la coupe. Il est possible de récupérer lôexception levée en paramétrant le code advice. Côest ce quôillustre le code suivant :

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

4.6 Introspection de points de jonction

Au fil de ce chapitre, nous avons vu des techniques permettant dôintroduire de la généricité dans les coupes. Il en résulte que les codes advices associés à une coupe peuvent

e lôapplication. Nous allons voir dans cette section les moyens fournis par AspectJ pour obtenir des informations sur le point de jonction actuel qui a provoqué lôexécution du code advice.

AspectJ introduit à cet effet un nouveau mot-clé : *thisJoinPoint*. A lôinstar de *this* qui est une référence à lôobjet courant, *thisJoinPoint* est une référence à un objet qui contient la description du point de jonction courant. Le type de *thisJoinPoint* est la classe **org.aspectj.lang.JoinPoint.** Le tableau suivant reprend les méthodes offertes par cette classe :

Méthode	Description
Object getThis ()	*Retourne l'objet source du point de jonction.* Si le point de jonction concerne une méthode, lôobjet source est lôobjet appelant.
Object getTarget ()	*Retourne l'objet cible du point de jonction.* Si le point de jonction concerne une méthode, lôobjet cible est lôobjet appelé.
Object [ ] getArgs ()	*Retourne les arguments du point de jonction.* Si le point de jonction concerne une méthode, les arguments sont les paramètres fournis à celle-ci.
String getKind ()	*Retourne le type du point de jonction sous forme de* **String.**
Signature getSignature ()	*Retourne la signature du point de jonction.* Par exemple, si le point de jonction concerne une méthode, lôinterface **Signature** permet de récupérer le nom, la visibilité, la classe et le type de retour de cette méthode.
SourceLocation getSourceLocation ()	*Retourne la localisation du point de jonction dans le code de l'application.* Lôinterface **SourceLocation** permet notamment de récupérer le nom de fichier et le numéro de ligne du point de jonction.

Imaginons, à titre dôexemple, un aspect de traçage imprimant le nom de chaque méthode appelée. Le code de cet aspect pourrait être le suivant :

```
package edu.ulb;
public aspect trace {
  before(): call(* *..*.*(..)) {
```

```
oinPoint.getSignature().getName());
        }
    }
```

Cet exemple illustre bien la puissance de la POA; une poignée de ligne de code suffit à décrire une fonctionnalité qui s'avère être très transversale (toutes les classes sont affectées).

4.7 Mécanisme d'introduction

Comme nous l'avons vu au point 3.6, il est possible d'étendre les classes d'une application en leur ajoutant divers éléments. AspectJ permet l'ajout de cinq types d'éléments : attribut, méthode, constructeur, classe héritée et interface implémentée

L'introduction de ces éléments se déclare à l'intérieur d'un aspect. En réalité, l'aspect déclare des éléments pour le compte des classes. C'est ce que l'on dénomme une déclaration intertype. Le programmeur doit toutefois faire attention à ne pas introduire un élément déjà existant dans la classe visée sous peine de voir la compilation échouer.

Bien évidemment, les éléments introduits ne peuvent être utilisés que par des aspects. L'application n'est pas supposée en connaître l'existence.

Cependant, il est quand même possible d'utiliser certains de ces éléments grâce au package **java.lang.reflect.** En effet, il est par exemple possible de récupérer toutes les méthodes d'une classe au *runtime*. Les méthodes introduites se retrouvent alors dans l'ensemble obtenu.

Nous allons maintenant passer en revue les différents éléments introductibles.

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 :

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

b) Méthode

le à lôintroduction dôattribut. Elle se réalise comme la définition dôune méthode au sein dôune classe, sauf que lôon ajoute comme préfixe le nom de la classe cible.

```
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 dôintroduction dôAspectJ, un constructeur est juste une méthode dont le nom est **new**.

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

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

d) Classe héritée

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

```
public aspect heritage {
        declare parents: Class1 extends Class2;
}
```

Dans lôexemple précédent, nous avons défini que *Class1* hérite désormais de *Class2*. Imaginons que Class1 héritait déjà de *Class3,* alors lôintroduction nôest possible que si *Class2* soit une sous-classe de *Class3*.

Le nom de la classe à modifier peut contenir des wildcards, ce qui rend possible la modification de la hiérarchie dôun ensemble de classe.

4.8 Interface implémentée

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 à toutes les classes dont le nom contient **foo**.

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

4.9 Héritage d'aspects

A l'instar des classes, un mécanisme d'héritage est proposé par AspectJ.

Un aspect peut étendre un aspect abstrait ou une classe. Dans le second cas, l'aspect hérite des méthodes et des attributs définis dans la classe qu'il étend.

Un aspect abstrait se définit à l'aide du mot-clé *abstract* placé devant le mot-clé aspect. Un aspect abstrait n'est pas tissé mais permet de centraliser des codes advices communs à plusieurs aspects. Seules les coupes nommées peuvent être étendues dans un sous-aspect.

En surchargeant une coupe, un sous-aspect spécifie l'ensemble des points de jonction auxquels vont s'appliquer les codes advices liés à cette coupe dans l'aspect abstrait. Illustrons notre propos par un exemple :

```
public abstract aspect SuperAspect {
   abstract pointcut coupe();
   before(): coupe() { ... } //code advice 1
}
public aspect SubAspect extends SuperAspect {
   pointcut coupe(): call(* Class.*(..));
}
```

Dans cet exemple, *SubAspect* surcharge la coupe *coupe*, ce qui implique que le *code advice 1* sera exécuté à chaque appel de méthode de la classe Class.

4.10 Ordonnancement d'aspects

on soit associé à plusieurs aspects. Dans certains cas, lôordre dans lequel ces aspects sont exécutés est important. AspectJ fournit à cet effet le mot-clé *declare precedence :*

```
public aspect OrdreAspect {
  declare precedence: Aspect1, Aspect2;
}
```

Ce code indique que lôaspect Aspect1 doit être appliqué avant lôaspect *Aspect2*. Il est également possible de faire usage de wildcards dans les noms des aspects.

- Si aucune précision nôest fournie quant à lôordre dans lequel tisser les aspects, *AspectJ* applique les règles implicites suivantes :

1. Les codes advices définis dans un sous-aspect ont la priorité sur les codes advices hérités.
2. Pour tout couple de codes advices défini dans le même aspect :

ï Si lôun des deux codes advices est de type *after*, celui qui est défini en deuxième a la priorité.

ï Sinon, le code advice défini en premier a la priorité.

3. Lôordre de tissage nôest pas spécifié pour deux codes advices définis dans deux aspects non liés par une relation dôhéritage.

Ces règles ne sont pas encore parfaites, elles peuvent en effet mener dans certains cas à des cycles de priorité. Dans lôexemple suivant, le deuxième code advice est prioritaire sur le premier, le troisième est prioritaire sur le second et le premier emporte la priorité sur le troisième.

```
public aspect ordre {
    before(): execution(void main(String[] args)) { ... }
    after(): execution(void main(String[] args)) { ... }
    before(): execution(void main(String[] args)) { ... }
}
```

5. Analyse des approches existantes pour la modélisation par aspect

Si les termes de programmation par aspect et de séparation avancée des préoccupations (Advanced Separation of Concerns) [36,37], sont de plus en plus largement utilisés, la plupart des enjeux relatifs à lôanalyse et à la conception par aspects relevés lors de leur introduction, notamment dans [38], sont encore dôactualité de nos jours : quôest ce que

? Comment identifier les aspects au niveau de ces phases préalables à la programmation ? Etc.

Plusieurs travaux récemment proposés se sont ainsi intéressés à la séparation des préoccupations transversales (aspects) tout au long du cycle de développement, et en particulier au niveau de la phase de conception. Nous présentons et discutons ci-dessous la plupart de ces approches ou propositions de modélisation, dont les principales caractéristiques sont résumées dans le tableau1.1. La synthèse de ces propositions est basée sur les propriétés suivantes :

- *Portée.* La portée d'une proposition détermine la phase du cycle de développement considérée par celle-ci : la phase d'analyse et de recensement des besoins (analysis), ou la phase de conception (design).

- *Extension.* La plupart des approches existantes proposent, pour la définition de leurs concepts et relations Aspect, d'étendre le métamodèle d'UML. Elles se distinguent cependant par leurs façons de faire : extension par stéréotypage (stereotyping), par méta modélisation (metamodeling), ou proposition d'un profile UML.

- *Vues.* Pour représenter un système à base d'aspects il convient le plus souvent de s'intéresser non seulement à la modélisation de sa structure statique, mais aussi à la modélisation de sa partie dynamique et donc à l'expression du résultat du tissage des aspects. Une proposition peut s'intéresser exclusivement à la vue structurelle (structural) ou comportementale (behavioural) d'un système, comme elle peut considérer les deux vues.

- *Notation graphique.* Les travaux actuels se concentrent le plus souvent sur la définition de la sémantique précise des concepts et relations qu'ils proposent. Il existe cependant des propositions qui offrent en plus des syntaxes concrètes pour la représentation graphique de leurs éléments de modélisation par aspects. Dans le tableau, la colonne correspondante à cette propriété indique si l'approche considérée propose une syntaxe concrète ou non.

[illegible]	[illegible]	Extension	Vues	Notation graphique
[30]	Conception	Méta modélisation et stéréotypage	Structurelle	Oui
[33]	Conception	Méta modélisation	Structurelle	Oui
[32]	Conception	Stéréotypage	Structurelle	Oui
[21]	Conception	Profil UML	Structurelle et comportementale	Oui
[25]	Conception	Méta modélisation	Structurelle	Oui

Tableau 1.1. Propriétés caractéristiques des principales approches de modélisation des aspects [24]

De manière générale lôanalyse des travaux évalués dans le tableau montre que :

- Toutes les propositions de modélisation par aspects sont dédiées à la phase de conception. De manière générale, ces propositions se concentrent sur la représentation des structures statiques des systèmes, mais très rarement sur la représentation de leurs comportements.

- UML, le langage standard de modélisation par objets, est en général considéré comme un point de départ à la définition dôun langage de modélisation par aspects, car il sôagit dôun langage standard et générique qui peut être facilement étendu et adapté.

- Deux approches dôextension dôUML sont largement utilisées : stéréotypage ou méta modélisation. Toutefois, bien que ces deux techniques soient complémentaires, les propositions de modélisation actuelles choisissent, en général, lôune ou lôautre de ces deux techniques en raison de leurs avantages et inconvénients. Nous pensons cependant quôune combinaison de ces deux techniques permet dôobtenir une extension dôUML pour les aspects qui soit plus avantageuse.

6. Évaluation des méthodes architecturales

Évaluer des méthodes de développement architectural consiste à les comparer, pour cela, nous disposons des plateformes d'évaluation.

L'une d'elles est la plateforme NIMSAD. Elle est très connue et date de 1986 (voir [26]). Cette plateforme est assez générale et permet de comparer toute sorte de méthodes de (problem solving).

Ali Babar propose la plateforme FOCSAAM [22] qui est spécifique pour comparer les méthodes d'analyse d'architecture.

et nous utilisons la plateforme FOCSAAM dans la suite de ce document pour évaluer notre méthode architecturale.

6.1 La plateforme NIMSAD: Normative Information Model based Systems Analysis and Design

La plateforme NIMSAD est décrite dans [26]. Cette plateforme est utilisée pour comprendre, analyser et comparer tous les catégories de méthodes de développement y compris les méthodes architecturales.

Cette plateforme a quatre éléments : le premier permet de décrire la situation du problème, le deuxième la solution prévue pour le résoudre, le troisième permet de décrire le processus de développement (problem-solving) et la quatrième permet dévaluer le processus de développement. La figure 1.6 montre ces éléments.

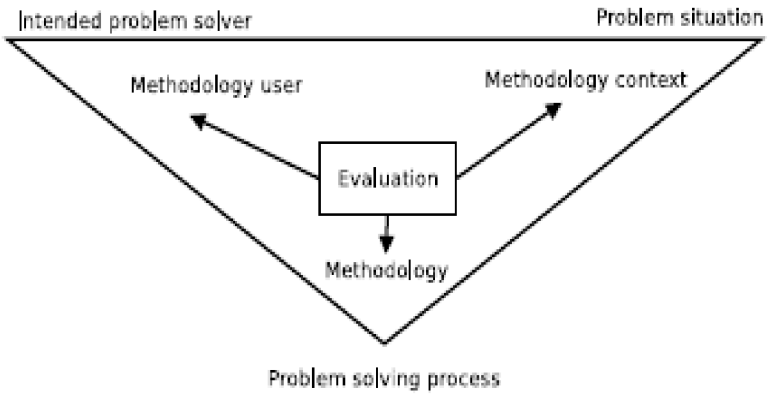


Fig.1.6 - Quatre différents éléments de NIMSAD pour évaluer les méthodes de développement **[26]**

6.2 La plateforme FOCSAAM : Framework for Comparing Software Architecture Analysis Methods

La plateforme FOCSAAM (Framework for Comparing Software Architecture Analysis Methods) [22]. FOCSAAM aide des architectes et des chefs de projets à choisir une méthode spécifique de développement d'architecture. FOCSAAM possède dix-sept éléments, qui sont organisés en quatre composants. Les composants de FOCSAAM correspondent aux quatre éléments de la plateforme NIMSAD. Cependant, à la différence de NIMSAD, FOCSAAM ne traite que les processus d'évaluation d'architecture logiciel. Ses dix sept éléments permettent de caractériser finement des méthodes d'évaluation d'architecture de logiciel. Elle permet d'aider à sélectionner une méthode de développement approprié à un contexte particulier. Elle

dix-sept éléments. A chaque élément correspond une question.

Ces questions sont décrites dans le tableau 1.2.

Composant	Éléments	Description succincte
Contexte	Définition de l'architecture	La méthode considère-t-elle explicitement une définition particulière de l'architecture du système ?
	Objectif de méthode	Quel est l'objective spécifique de méthode ?
	Attributs de qualités	La méthode couvre combien et les quels attributs de qualité ?
	Étapes applicables	Quelle est la phase de développement la plus appropriée pour appliquer la méthode ?
	Entrée/Sortie	Quelles sont les entrées exigées et les sorties produites ?
	Domaine d'application	Quel est le domaine d'application de la méthode en particulier ?
Donneurs d'ordre	Bénéfices	Quels sont les avantages pour les intervenants ?
	Intervenants	Qui participe à l'évaluation ?
	support de processus	Combien de processus est fourni par la méthode pour exécuter de diverses activités ?
	Aspects sociaux	Comment la méthode manipule-t-elle les aspects non technique (e.g. les issues sociales et d'organisation) ?
	Ressources exigées	Combien de journées-homme sont exigées ? Quelle est la taille de l'équipe d'évaluation ?
Contenus	Activités de la méthode	Quelles sont les activités à exécuter et dans quel ordre pour réaliser les buts ?
	Description architecturale	Quelle forme de description architecturale est recommandée (e.g. ADL, vues formels/ informels, etc.) ?
	Approches d'évaluation	Quel type d'approche d'évaluation est employé par la méthode ?
	Outil de support	Y a-t-il les outils ou le dépôt d'expérience pour soutenir la méthode et ses artefacts ?
Fiabilité	Niveau de maturité	Quel est le niveau de la fiabilité (commencement, développement, raffinement, etc.)
	Validation de méthode	La méthode a-t-elle été validée ? Comment a-t- elle été validée ?

***TAB.1.2 Composants et les éléments de la plateforme FOCSAAM et les questions d'évaluation* [22].**

Nous avons présenté dans ce chapitre les principaux concepts de la programmation orientée aspect, à savoir, les points de jonction, les coupes, les codes advice, et enfin le mécanisme d'introduction. Ensuite nous avons vu comment les concepts de la POA sont mis en pratique avec le langage *AspectJ*.

La programmation orientée aspect est une nouvelle méthodologie qui permet de séparer les préoccupations subsidiaires qui entrecoupent les fonctionnalités principales d'un système.

Plusieurs équipes de recherche ont travaillé à l'élaboration de langages de programmation pour le paradigme aspect (Aspect Oriented Programming). Toutefois, la proposition d'une nouvelle méthodologie de développement doit inclure non seulement la mise au point d'une méthode de programmation, mais également celle de méthodes spécifiques pour l'analyse et de conception. En effet, il existe une telle approche systématique appelée l'approche dirigée par les cas d'utilisation, elle fournit une technique solide pour développer des applications qui répondent bien aux attentes des utilisateurs. Ce sujet fera l'objet d'une étude plus approfondie dans le prochain chapitre.

Ainsi, dans ce projet nous nous focalisons sur la définition d'une méthode d'analyse et de conception orientée aspect et basée sur la notation UML. Nous pensons que l'adoption de l'orienté aspect va améliorer efficacement le développement tout en se basant sur l'approche dirigée par les cas d'utilisation d'UML.

Analyse des préoccupations et les cas d'utilisation

Ce chapitre couvre

- *Introduction*
- *Diagramme de cas d'utilisation*
- *Modélisation des besoins avec UML*
- *Cas d'utilisation et leurs propriétés*
- *Introduction à OCL*
- *Avantages du cas d'utilisation*
- *Besoins fonctionnels et non fonctionnels*
- *Conclusion*

Analyse des préoccupations et les cas dâutilisation

1. Introduction

Souvent, le développeur ne voit que lâutilité des diagrammes de classes, car ces derniers permettent une écriture immédiate du code associé; les autres diagrammes de UML paraissent peu utiles du fait quâils ne produisent pas de résultat tangible. Car tous ces diagrammes permettent dâacquérir une meilleure compréhension du système à construire; parmi ces diagrammes "inutiles", le plus important est le diagramme de cas dâutilisation.
Chaque fois que le développeur se trouve confronté à un dilemme, à quelque stade du développement que ce soit, il devrait penser aux diagrammes des cas dâutilisation. Plus simplement, il sâagit de se poser la question ***"Comment ferait l'utilisateur dans telle ou telle situation ?*** ␣

Lâutilisateur doit en tous temps rester satisfait, et si un système est conçu avec cette théorie, alors ce système sera forcément adéquat. Un système adéquat fera forcément un utilisateur satisfait, et lâutilisateur étant souvent également un client, un client satisfait est une subsistance suffisamment précieuse pour que lôon essaye par tous les moyens de la conserver en lôétat. Cêest le but vers lequel les cas dâutilisation cherchent à tendre. Les cas dâutilisation ne constituent pas à proprement parler une technique de programmation : il sâagit dâune attitude vis-à- vis dâun problème.

Dans ce chapitre nous allons concentrer notre étude sur la technique des cas dâutilisation et son utilité dans le développement de logiciels et dans la capture des différentes préoccupations des utilisateurs. Les cas dâutilisation présentent un bon moyen pour la spécification des besoins mais également ils présentent une meilleure technique pour lâingénierie des systèmes informatiques. Tout développement devrait être dirigé par les cas dâutilisation qui explosent bien les exigences des utilisateurs.

2. Diagramme de cas d'utilisation

ne sont pas des informaticiens. Il leur faut donc un
s. Côest précisément le rôle des diagrammes de cas dôutilisation qui permettent de recueillir, dôanalyser et dôorganiser les besoins, et de recenser les grandes fonctionnalités dôun système [15]. Il sôagit donc de la première étape UML dôanalyse dôun système.

Un diagramme de cas dôutilisation capture le comportement dôun système, dôun sous-système, dôune classe ou dôun composant tel quôun utilisateur extérieur le voit. Il scinde la fonctionnalité du système en unités cohérentes, les cas dôutilisation, ayant un sens pour les acteurs. Les cas dôutilisation (CU) permettent dôexprimer le besoin des utilisateurs dôun système, ils sont donc une vision orientée utilisateur de ce besoin au contraire dôune vision informatique.

Étant donné lôimportance des CU dans la majorité des approches qui emploient *Unified Modeling Language* (UML) et la diffusion de cette notation, nous allons extraire la définition de son manuel de référence [1] : « *description d'un ensemble de séquences d'actions incluant des variantes qu'un système exécute* ».

Les trois citations suivantes, tirées de livres très populaires en génie logicielle, montrent que le syntagme « séquences dôactions » de la définition précédente est, en quelques sorte, un synonyme dôexigence fonctionnelle :

- Sommerville [2] : « *les élicitations fondées sur les CU sont toujours plus employées pour l'élicitation des exigences* fonctionnelles » ;
- Ghezzi [3] : « *Les diagrammes CU décrivent le contexte global d'un système en subdivisant les fonctionnalités du système en transactions qui sont utiles pour les acteurs et qui montrent comment les acteurs interagissent avec eux.* »
- Larman [4] « *CU* sont *des exigences fonctionnelles qui indiquent ce que le système doit faire* »

2.1 Éléments des diagrammes de cas d'utilisation

Les cas dôutilisation énumèrent toutes les interactions possibles entre de système et son environnement extérieur. A cet effet, un diagramme de cas dôutilisation est constitué des éléments suivants :

2.1.1 Acteur

Un acteur est lôidéalisation dôun rôle joué par une personne externe, un processus ou une chose qui interagit avec un système. [10]

me (figure 2.1) avec son nom (i.e. son rôle) inscrit

Il est également possible de représenter un acteur sous la forme dôun classeur stéréotypé « actor ». (figure 2.2).

Fig.2.1 - Exemple de représentation d'un acteur **[19]**

Fig.2. 2 – Exemple de représentation d'un acteur sous la forme d'un classeur **[19]**

2.1.2 Cas d'utilisation

Un cas dôutilisation est une unité cohérente représentant une fonctionnalité visible de lôextérieur. [10]

Il réalise un service de bout en bout, avec un déclenchement, un déroulement et une fin, pour lôacteur qui lôinitie. Un cas dôutilisation modélise donc un service rendu par le système, sans imposer le mode de réalisation de ce service.

Un cas dôutilisation se représente par une ellipse (figure 2 .3) contenant le nom du cas (un verbe à lôinfinitif), et optionnellement, au-dessus du nom, un stéréotype.

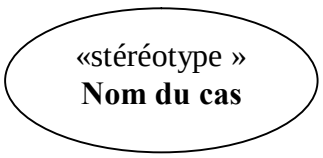


Fig.2.3 – Exemple de représentation d'un cas d'utilisation **[19]**

Dans le cas où lôon désire présenter les attributs ou les opérations du cas dôutilisation, il est préférable de le représenter sous la forme dôun classeur stéréotypé « use case » (figure 2 .4).

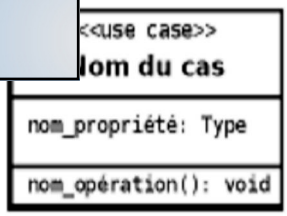


Fig.2.4 – Exemple de représentation d'un cas d'utilisation sous la forme d'un classeur **[19]**

2.1.3 Représentation d'un diagramme de cas d'utilisation

Comme le montre la figure 2.5, la frontière du système est représentée par un cadre. Le nom du système figure à lôintérieur du cadre, en haut. Les acteurs sont à lôextérieur et les cas dôutilisation à lôintérieur.

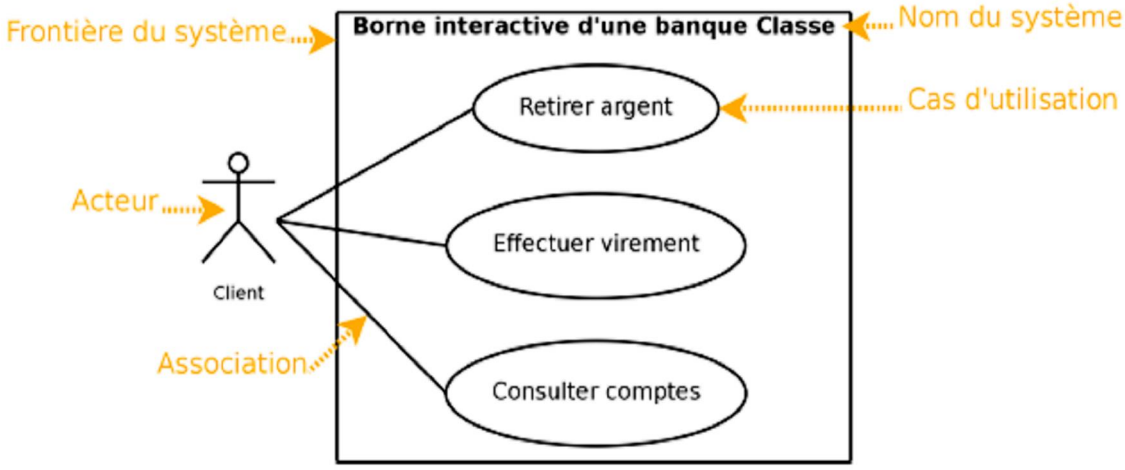


Fig.2.5 – Exemple simplifié de diagramme de cas d'utilisation modélisant une borne d'accès à une banque **[19].**

2.2 Relations dans les diagrammes de cas d'utilisation

Dans un diagramme de cas dôutilisation, on trouve trois types de relations, ces dernières sont décrites dans les sections qui suivent :

2.2.1 Relations entre acteurs et cas d'utilisation

❖ **Relation d'association**

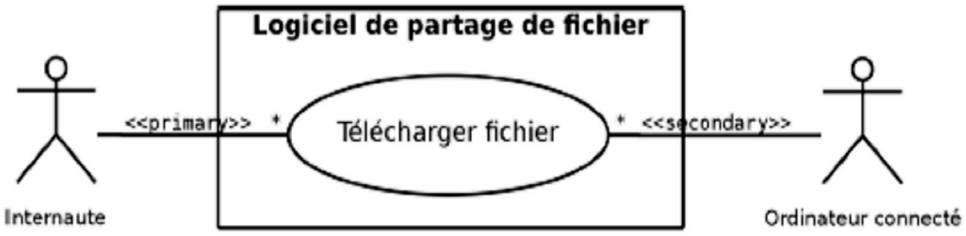


Fig.2. 6 – Diagramme de cas d'utilisation représentant un logiciel de partage de fichiers **[19]**

e communication entre un acteur et un cas dôutilisation
f. figure 2.5 ou 2.6).

❖ **Multiplicité**

Lorsquôun acteur peut interagir plusieurs fois avec un cas dôutilisation, il est possible dôajouter une multiplicité sur lôassociation du côté du cas dôutilisation [11]. Le symbole * signifie plusieurs (figure 2.6), exactement n sôécrit tout simplement n, n..m signifie entre n et m, etc. Préciser une multiplicité sur une relation nôimplique pas nécessairement que les cas sont utilisés en même temps.

❖ **Acteurs principaux et secondaires**

Un acteur est qualifié de principal pour un cas dôutilisation lorsque ce cas rend service à cet acteur. Les autres acteurs sont alors qualifiés de secondaires. Un cas dôutilisation a au plus un acteur principal. Un acteur principal obtient un résultat observable du système tandis quôun acteur secondaire est sollicité pour des informations complémentaires. En général, lôacteur principal initie le cas dôutilisation par ses sollicitations. Le stéréotype « primary » vient orner lôassociation reliant un cas dôutilisation à son acteur principal, le stéréotype « secondary » est utilisé pour les acteurs secondaires (figure 2.6).

❖ **Cas d'utilisation interne**

Quand un cas nôest pas directement relié à un acteur, il est qualifié de cas dôutilisation interne.

2.2.2 Relations entre cas d'utilisation

Les cas dôutilisation ne génèrent pas de code, contrairement aux diagrammes de classes. En revanche, les cas dôutilisation peuvent avoir des relations entre eux.

❖ **Types et représentations**

Il existe principalement deux types de relations :

ï les dépendances stéréotypées, qui sont explicitées par un stéréotype (les plus utilisés sont lôinclusion et lôextension),

ï et la généralisation/spécialisation.

Une dépendance se représente par une flèche avec un trait pointillé (figure 2.7). Si le cas A inclut ou étend le cas B, la flèche est dirigée de A vers B.

Le symbole utilisé pour la généralisation est une flèche avec un trait plein dont la pointe est un triangle fermé désignant le cas le plus général (figure 2.7).

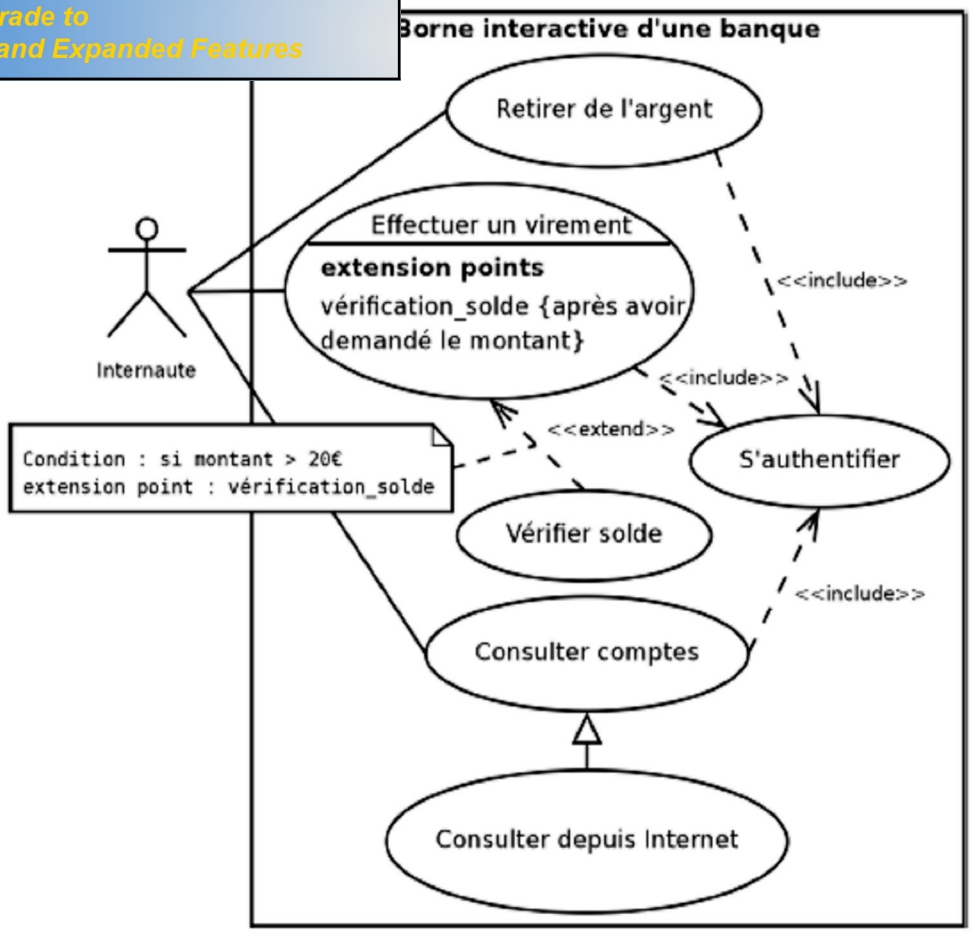


Fig.2.7 – Exemple de diagramme de cas d'utilisation **[19]**

a) Relation d'inclusion

Un cas A inclut un cas B si le comportement décrit par le cas A inclut le comportement du cas B [12] : le cas A dépend de B. Lorsque A est sollicité, B lôest obligatoirement, comme une partie de A.

Cette dépendance est symbolisée par le stéréotype « include » (figure 2.7). Par exemple, lôaccès aux informations dôun compte bancaire inclut nécessairement une phase dôauthentification avec un identifiant et un mot de passe (figure 2.7).

Les inclusions permettent essentiellement de factoriser une partie de la description dôun cas dôutilisation qui serait commune à dôautres cas dôutilisation (cf. le cas Sôauthentifier de la figure 2.7).

Les inclusions permettent également de décomposer un cas complexe en sous-cas plus simples (figure 2.8). Cependant, il ne faut surtout pas abuser de ce type de décomposition : il faut éviter de réaliser du découpage fonctionnel dôun cas dôutilisation en plusieurs sous-cas dôutilisation pour ne pas retomber dans le travers de la décomposition fonctionnelle.

bas, puisquôil nôy a aucune représentation temporelle

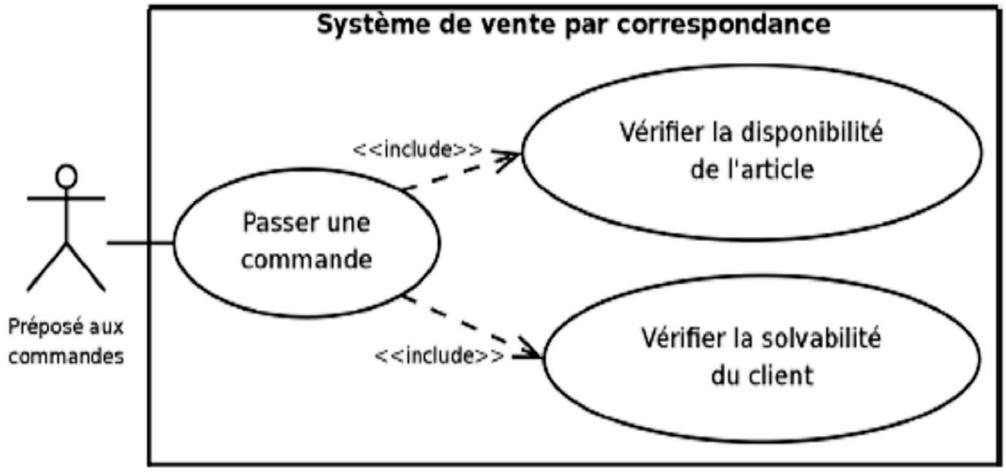


Fig.2.8 – Relations entre cas pour décomposer un cas complexe **[19]**

b) Relation dôextension

La relation dôextension est probablement la plus utile car elle a une sémantique qui a un sens du point de vue métier au contraire des deux autres qui sont plus des artifices dôinformaticiens.

On dit quôun cas dôutilisation A étend un cas dôutilisation B lorsque le cas dôutilisation A peut être appelé au cours de lôexécution du cas dôutilisation B [12]. Exécuter B peut éventuellement entraîner lôexécution de A : contrairement à lôinclusion, lôextension est optionnelle. Cette dépendance est symbolisée par le stéréotype « extend » (figure 2.7).

Lôextension peut intervenir à un point précis du cas étendu. Ce point sôappelle le *point d'extension*. Il porte un nom, qui figure dans un compartiment du cas étendu sous la rubrique point dôextension, et est éventuellement associé à une contrainte indiquant le moment où lôextension intervient. Une extension est souvent soumise à condition. Graphiquement, la condition est exprimée sous la forme dôune note. La figure 2.7 présente lôexemple dôune banque où la vérification du solde du compte nôintervient que si la demande de retrait dépasse 20 euros.

c) Relation de généralisation

Un cas A est une généralisation dôun cas B si B est un cas particulier de A [12]. Dans la figure 2.7, la consultation dôun compte via Internet est un cas particulier de la consultation. Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept dôhéritage dans les langages orientés objet.

acteurs est la généralisation : un acteur A est une généralisation dôun acteur B si lôacteur A peut être substitué par lôacteur B [12]. Dans ce cas, tous les cas dôutilisation accessibles à A le sont aussi à B, mais lôinverse nôest pas vrai.

Le symbole utilisé pour la généralisation entre acteurs est une flèche avec un trait plein dont la pointe est un triangle fermé désignant lôacteur le plus général (comme nous lôavons déjà vu pour la relation de généralisation entre cas dôutilisation).

Par exemple, la figure 2.9 montre que le directeur des ventes est un préposé aux commandes avec un pouvoir supplémentaire : en plus de pouvoir passer et suivre une commande, il peut gérer le stock. Par contre, le préposé aux commandes ne peut pas gérer le stock.

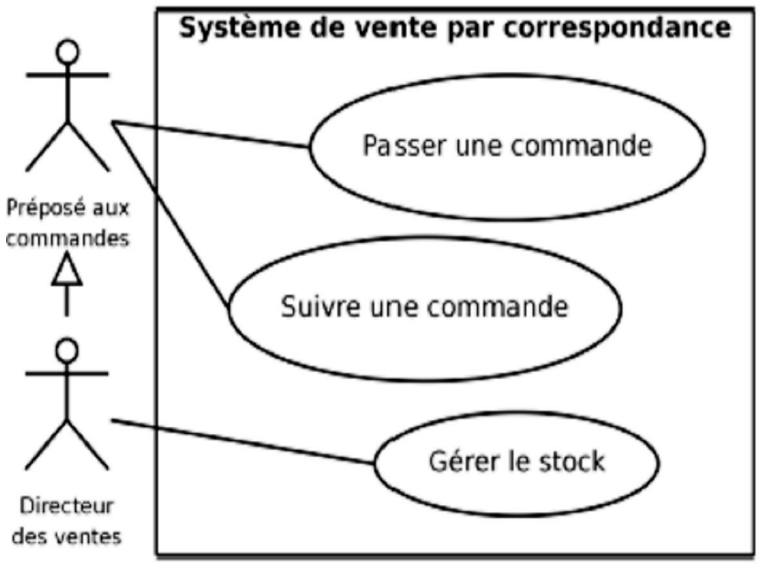


Fig.2.9 – Relations entre acteurs **[19]**

3. Modélisation des besoins avec UML

Cette section traite du rôle que tient UML pour compléter la capture des besoins. La technique des cas dôutilisation est la pierre angulaire de cette étape. Noua verrons successivement dans cette section comment identifier les acteurs, recenser et décrire les cas dôutilisation.

3.1 Identification des acteurs

UML nôemploie pas le terme dôutilisateur mais dôacteur. Les acteurs dôun système sont les entités externes à ce système qui interagissent (saisie de données, réception

s sont donc à lôextérieur du système et dialoguent avec

r lôinterface que le système va devoir offrir à son environnement. Oublier des acteurs ou en identifier de faux conduit donc nécessairement à se tromper sur lôinterface et donc la définition du système à produire.

Il faut faire attention à ne pas confondre acteurs et utilisateurs (utilisateur avec le sens de la personne physique qui va appuyer sur un bouton) dôun système. Dôune part parce que les acteurs incluent les utilisateurs humains mais aussi les autres systèmes informatiques ou hardware qui vont communiquer avec le système. Dôautre part parce que un acteur englobe toute une classe dôutilisateur. Ainsi, plusieurs utilisateurs peuvent avoir le même rôle, et donc correspondre à un même acteur, et une même personne physique peut jouer des rôles différents vis-à-vis du système, et donc correspondre à plusieurs acteurs.

Chaque acteur doit être nommé. Ce nom doit refléter sont rôle car un acteur représente un ensemble cohérent de rôles joués vis-à-vis du système.

Pour trouver les acteurs dôun système, nous devons identifier quels sont les différents rôles que vont devoir jouer ses utilisateurs (ex : responsable clientèle, responsable dôagence, administrateur, approbateur, . . .). Il faut également sôintéresser aux autres systèmes avec lesquels le système va devoir communiquer comme :

ï les périphériques manipulés par le système (imprimantes, hardware dôun distributeur de billet, . . .) ;

ï des logiciels déjà disponibles à intégrer dans le projet ;

ï des systèmes informatiques externes au système mais qui interagissent avec lui, etc.

Pour faciliter la recherche des acteurs, on peut imaginer les frontières du système. Tout ce qui est à lôextérieur et qui interagit avec le système est un acteur, tout ce qui est à lôintérieur est une fonctionnalité à réaliser.

Il faut vérifier que les acteurs communiquent bien directement avec le système par émission ou réception de messages. Une erreur fréquente consiste à répertorier en tant quôacteur des entités externes qui nôinteragissent pas directement avec le système, mais uniquement par le biais dôun des véritables acteurs. Par exemple, lôhôtesse de caisse dôun magasin de grande distribution est un acteur pour la caisse enregistreuse, par contre, les clients du magasin ne correspondent pas à un acteur car ils nôinteragissent pas directement avec la caisse.

ation

écrire exhaustivement les exigences fonctionnelles du système. Chaque cas dôutilisation correspond donc à une fonction métier du système, selon le point de vue dôun de ses acteurs. Aussi, pour identifier les cas dôutilisation, il faut se placer du point de vue de chaque acteur et déterminer comment et surtout pourquoi il se sert du système. Il faut éviter les redondances et limiter le nombre de cas en se situant à un bon niveau dôabstraction.

Nommer les cas dôutilisation avec un verbe à lôinfinitif suivi dôun complément tout en se plaçant du point de vue de lôacteur et non pas de celui du système. Par exemple, un distributeur de billets aura probablement un cas dôutilisation Retirer de lôargent et non pas Distribuer de lôargent.

De par la nature fonctionnelle, et non objet, des cas dôutilisation, et en raison de la difficulté de trouver le bon niveau de détail, il faut être très vigilant pour ne pas retomber dans une décomposition fonctionnelle descendante hiérarchique. Un nombre trop important de cas dôutilisation est en général le symptôme de ce type dôerreur.

Dans tous les cas, il faut bien garder à lôesprit quôil nôy a pas de notion temporelle dans un diagramme de cas dôutilisation.

3.3 Description textuelle des cas d'utilisation

Le diagramme de cas dôutilisation décrit les grandes fonctions dôun système du point de vue des acteurs, mais nôexpose pas de façon détaillée le dialogue entre les acteurs et les cas dôutilisation. Bien que de nombreux diagrammes dôUML permettent de décrire un cas, il est recommandé de rédiger une description textuelle car côest une forme souple qui convient dans bien des situations.

Une description textuelle couramment utilisée se compose de trois parties [13].

1. La première partie permet dôidentifier le cas, elle doit contenir les informations qui suivent.
   - **Nom :** Utiliser une tournure à lôinfinitif (ex : Réceptionner un colis).
   - **Objectif :** Une description résumée permettant de comprendre lôintention principale du cas dôutilisation. Cette partie est souvent renseignée au début du projet dans la phase de découverte des cas dôutilisation.

ui vont réaliser le cas dôutilisation (la relation avec le
rée par le trait liant le cas dôutilisation et lôacteur dans un diagramme de cas dôutilisation)

- **Acteurs secondaires :** Ceux qui ne font que recevoir des informations à lôissue de la réalisation du cas dôutilisation
- **Dates :** Les dates de créations et de mise à jour de la description courante.
- **Responsable :** Le nom des responsables.
- **Version :** Le numéro de version.

2. La deuxième partie contient la description du fonctionnement du cas sous la forme dôune séquence de messages échangés entre les acteurs et le système. Elle contient toujours une séquence nominale qui décrit de déroulement normal du cas. À la séquence nominale sôajoutent fréquemment des séquences alternatives (des embranchements dans la séquence nominale) et des séquences dôexceptions (qui interviennent quand une erreur se produit).

- **Les pré conditions :** elles décrivent dans quel état doit être le système (lôapplication) avant que ce cas dôutilisation puisse être déclenché.
- **Des scénarii :** Ces scénarii sont décrits sous la forme dôéchanges dôévènements entre lôacteur et le système. On distingue le scénario nominal, qui se déroule quand il nôy a pas dôerreur, des scénarii alternatifs qui sont les variantes du scénario nominal et enfin les scénarii dôexception qui décrivent les cas dôerreurs.
- **Des post conditions :** Elle décrivent lôétat du système à lôissue des différents scénarii.

3. La troisième partie de la description dôun cas dôutilisation est une rubrique optionnelle. Elle contient généralement des spécifications non fonctionnelles (spécifications techniques, . . .). Elle peut éventuellement contenir une description des besoins en termes dôinterface graphique.

4. Cas d'utilisation et leurs propriétés

Cette section présente quelques remarques concernant les diagrammes de cas dôutilisation ainsi que leurs propriétés.

4.1 Relations dans les cas d'utilisation

Il est important de noter que lôutilisation des relations nôest pas primordiale dans la rédaction des cas dôutilisation et donc dans lôexpression du besoin. Ces relations peuvent être utiles dans certains cas mais une trop forte focalisation sur leur usage conduit souvent à une perte de temps ou à un usage faussé, pour une valeur ajoutée, au final, relativement faible.

cantonnés à lôingénierie des besoins, les diagrammes de cas dôutilisation ne peuvent être qualifiés de modélisation à proprement parler. Dôailleurs, de nombreux éléments descriptifs sont en langage naturel. De plus, ils ne correspondent pas stricto sensu à une approche objet. En effet, capturer les besoins, les découvrir, les réfuter, les consolider, etc., correspond plus à une analyse fonctionnelle classique.

5. Introduction à OCL

Le langage OCL (Object Constraint Language) [17] a été créé spécialement pour répondre au besoin de formaliser des contrats sur les modèles UML. OCL permet dôexprimer des contraintes restreignant les domaines de valeurs pouvant être ajoutées aux éléments du modèle UML.

Le langage OCL est fortement typé. Le langage consiste en un ensemble de types et dôopérations prédéfinis, regroupés et documentés sous la forme dôun package UML appelé UML_OCL. Le concepteur dôun modèle UML est censé importer ce package dans son modèle pour disposer des fonctionnalités dôOCL. Le package UML_OCL contient des types élémentaires tels les booléens, entiers, réels et chaîne de caractères, ainsi que des types plus complexes tels les types énumérés ou les collections.

Tous les types OCL sont des sous-types du type OclAny qui définit les propriétés communes à tous les objets OCL.

À ces types prédéfinis, viennent sôajouter des types « importés » qui correspondent aux classes, interfaces et type de données du modèle. En effet, une expression OCL est toujours écrite dans un contexte qui peut être soit un type, soit une opération, et qui sert de point dôorigine lorsque lôon veut naviguer dans un modèle. Si le contexte est un type, le mot-clé self dans une expression se rapporte à un objet de ce type. Si le contexte est une opération, self désigne le type qui possède cette opération. Tous les types OCL, quôils soient prédéfinis ou importés sont accessibles via une instance du type OCL OclType.

6. Avantages du cas d'utilisation

Les exigences non fonctionnelles sont souvent les exigences les plus contraignantes et les plus difficiles à satisfaire, mais on ne peut sôopposer à lôaffirmation que, sans une compréhension approfondie des fonctions, il est impossible de porter un projet à terme. Dans lôingénierie en général, et dans le Génie Logiciel en particulier, tout le monde sôentend aussi sur le fait que la façon de décrire (ou de spécifier) les fonctions et les méthodes et les outils employés sont très importants.

antages des CU » en [5], exprime, on ne peut plus s CU comme mécanisme de description des exigences fonctionnelles : « *La puissance de l'approche CU naît du fait qu'elle met au centre la tâche et l'utilisateur. Les utilisateurs auront des attentes plus claires de ce que le nouveau système leur permettra de faire que si vous prenez une approche mettant au centre les fonctions. Les CU aident les analystes et les développeurs à comprendre les règles d'affaires et le domaine d'application.* »

Parmi les faiblesses quôon peut trouver pendant lôanalyse des besoins [7, 8,9]:

- Il est parfois difficile de différencier ce que fait le système par rapport à ce que font les acteurs ;
- Les échanges entre le système et les acteurs sont souvent mal formalisés ;
- La différence entre les rôles des utilisateurs nôest pas toujours claire ;
- Les utilisateurs ont des difficultés à valider les fonctions, car les liens avec leurs actions ne sont pas toujours évidents.
- Les fonctions sont souvent trop abstraites.

Ces faiblesses sont pratiquement réglées avec une bonne application des CU, il serait plus correct de dire que les CU ont été introduits pour régler ces faiblesses. Voici une synthèse des avantages des CU en fonction des activités concernant les exigences.

- *Élicitation.* Le but de cette activité étant de comprendre ce dont les utilisateurs et les clients ont besoin, il est plus facilement atteint si on permet à lôutilisateur de raconter le problème de lôinteraction qui a eu lieu ou qui a été envisagée avec le système. Le problème, est par la suite transformé en CU.
- *Catégorisation*. Dans cette activité, il sôagit de catégoriser les exigences selon des dimensions qui faciliteront la mise en oeuvre : nécessité, priorité, stabilité, portée, etc.

Le fait que lôon puisse dialoguer avec la bonne personne (celle qui est concernée par le CU et qui a donné le contenu à insérer dans le CU) rend la tâche aisée.

- *Modélisation conceptuelle*. Puisque chaque « acteur » parle de sa propre expérience concrète liée aux actions quôil exécute, il est plus facile de mettre en évidence les concepts importants (pour lôutilisateur). Ce qui rend la compréhension et la validation plus faciles.

« lôacteur » qui est dans le scénario est celui-là même
laires à propos des éléments avec lesquels il interagit.

Et sôil ne sait pas très bien si côest le matériel ou le logiciel, le sous-système n° 1 ou le sous-système n° 2 qui doit lui répondre, il est facile de lui poser les bonnes questions.

- *Résolution de conflits*. Les conflits pour un rôle donné sont facilement réglables en questionnant en profondeur une personne et/ou en réalisant des entretiens avec dôautres intervenants qui peuvent jouer le même rôle. Les conflits entre rôles sont identifiés par les gestionnaires ou les responsables du système.
- *Spécification.* Toutes les considérations que nous venons de faire facilitent la rédaction des principes dôopération et des spécifications des exigences.
- *Validation des exigences.* Chaque acteur peut facilement valider ses CU et les gestionnaires ou les responsables systèmes peuvent valider lôensemble, mais en sachant que chaque vue est, en principe, valide.

Mais ce nôest pas seulement dans le domaine de lôingénierie des exigences que les CU sont des éléments utiles et efficaces. Ils facilitent aussi la conception système, les tests, la formation et la planification.

- *Conception système*. Lôimmédiateté du passage des CU aux diagrammes de séquences système facilite lôécriture et la formalisation des réponses du système.
- *Tests.* Chaque action peut être considérée comme un élément de la conception des tests ayant un ensemble fonctionnellement cohérent de cas à tester. Comme lôécrit Perry William en [6] « *En introduisant les CU dans le cycle de développement, on fait plus facilement face aux cas de tests incorrects, incomplets et manquants*. *L'emploi d'une approche fondée sur les CU assure non seulement que l'on satisfasse les exigences mais aussi les attentes* ».
- *Formation.* La structuration par rôles favorise la production dôun matériel didactique orienté vers les caractéristiques des utilisateurs et une organisation de lôapprentissage et de lôenseignement par problèmes.
- *Planification.* Les CU, avec leur organisation en actions discrètes, sont lôun des intrants les plus sûrs et faciles à quantifier pour lôestimation des coûts des projets sôinscrivant dans une approche par points de fonction.

les CU dépendent du fait quôils forcent les parties rfaces) entre les acteurs et le système et le contexte fonctionnel dans lequel le système opère dès les débuts du cycle de vie. Côest cette approche à boîte fermée renforcée par une mise en évidence des rôles et des séquences dôactions qui, en permettant de fractionner fonctionnellement le système, le rend plus facile à comprendre et à réaliser.

Les CU [7] sont des méthodes/outils dôélicitation et dôanalyse qui facilitent lôobtention de certains niveaux de qualité sinon pour toutes les caractéristiques présentées en [8] au moins pour lôaptitude, lôexactitude, la capacité fonctionnelle et la facilité de test.

Trois derniers points pour les CU avant de terminer cette section :

1. Même si certains auteurs opposent une approche Stimulus-Réponse (S-R) aux CU il est clair que les CU aussi sont une description des réponses causées par des stimuli. On pourrait dire que les CU sont une approche (S-R) organisée par rôles.
2. Les CU ne se limitent pas à une approche par objets même si leurs éclosions ont eu lieu à peu près en même temps.
3. Il nôest pas vrai que, pour les interactions entre les humains et la machine, les CU aient besoin de la définition des interfaces (fenêtres, champs, clavieré). Les CU, comme les fonctions « classiques », peuvent ð et doivent ð être organisés en une hiérarchie qui commence avec les intentions des acteurs.

7. Besoins fonctionnels et non-fonctionnels

De façon générale, les besoins ou exigences exprimés par les utilisateurs peuvent se classer en:

- Fonctionnel (ou dôapplication)
- Non-fonctionnel ou technique (ou dôinfrastructure)

Une propriété fonctionnelle traite un aspect particulier du comportement d'un système, elle est liée à une condition fonctionnelle bien définie. [31] définit :

“ La fonctionnalité est la capacité du système à accomplir des tâches prévues. Les besoins fonctionnels d'un système sont souvent définis avec des cas d'utilisations ”.

Une propriété non fonctionnelle, elle aussi, traite un aspect particulier du comportement d'un système, mais n'est liée à aucune condition fonctionnelle. C'est typiquement le cas des aspects qualitatifs du système. Dans le cas d'un système, [31] définit la qualité ainsi :

ré auquel ce système répond à ses exigences non
esoins fonctionnels".

La définition de la qualité a été normalisée par l'AFNOR dans ce sens :

"La qualité est l'aptitude d'un produit ou d'un service à répondre aux besoins exprimés d'un utilisateur", (voir [20]).

Les besoins fonctionnels sont modélisés sous forme de cas d'utilisation. Ici, fonction est à prendre au sens "fonctionnalité majeure" du système, ce que doit faire le système est par nature fonctionnel.

En ce qui concerne les besoins non-fonctinnels, UML offre ensuite des mécanismes d'extension tels que *étiquettes* et *contraintes*. Ces mécanismes sont pertinents pour spécifier des caractéristiques associées aux besoins telles que la sécurité, le login, temps de réponse, performance, etc.... dans notre approche, nous allons travailler sur la modélisation des propriétés non-fonctionnelles qui améliorent la qualité des logiciels à développer tout en se basant sur la technique des cas dôutilisation.

8. Conclusion

Nous avons présenté dans ce chapitre la technique dirigée par les cas dôutilisation pour la spécification des besoins des utilisateurs qui est lôune des phases architecturales les plus importantes. Ainsi, nous avons exposé quelques définitions dans ce domaine, et montré la place des cas dôutilisation dans le cycle de vie dôun logiciel et surtout pendant la phase dôanalyse des besoins.

L'étude de ce domaine nous a permis de constater qu'un des aspects importants de conception de logiciels est de satisfaire les propriétés non fonctionnelles du système, malgré cela la plupart de méthodes architecturales ne les prennent pas en charges.

Nous avons jugé nécessaire d'avoir une méthode de développement qui prend toutes les propriétés non fonctionnelles en charge et qui se laisse guider par ces propriétés. En plus, nous pensons que les cas dôutilisation facilitent le développement dans le sens où ils génèrent des solutions que les développeurs peuvent adapter aux besoins du système et que ces solutions ont déjà fait preuves d'être correctes. Un autre avantage d'utilisation des cas dôutilisation est la souplesse de développement.

ésenter notre contribution à savoir la proposition dôune
iciels fondée sur les propriétés non-fonctionelles considérées comme des aspects et dirigée par les cas dôutilisation.

3 Une méthode de développement de logiciels basée sur la notion d'aspect et orientée utilisateur

Ce chapitre couvre

- *Introduction*
- *Approche globale*
- *Motivations et objectifs*
- *Approche adoptée : modélisation des préoccupations transversales*
- *Processus de développement*
- *Conclusion*

Une méthode de développement de logiciels basée sur la notion dôaspect et orientée utilisateur

1. Introduction :

L'architecture logicielle d'un système permet de prendre en compte outre les besoins fonctionnels, les objectifs de qualité ou besoins non fonctionnels du système à développer. La prise en compte dès le début, dans le cycle de vie d'un système, de ces aspects est essentielle pour le développement des systèmes de qualité. La décomposition fonctionnelle d'un système en général est réalisée en utilisant des méthodes classiques de développement de logiciel; cependant il n'y a aucune méthode fiable considérant la décomposition non fonctionnelle. Dans ce travail nous proposons une méthode de développement qui garantit la prise en compte des propriétés de qualité. L'architecture se définit en considérant d'une part une spécification des propriétés non fonctionnelles du domaine qui seront considérées comme des aspects et d'autre part les besoins fonctionnels du système définis par des cas dôutilisation (vision orientée utilisateur). La configuration initiale est raffinée en introduisant des nouveaux éléments, répondant aux propriétés non fonctionnelles spécifiques. Notre contribution réside dans le fait de rendre explicites toutes les décisions architecturales, y compris celles concernant la prise en compte des besoins non fonctionnels.

2. Motivations et objectifs

Puisque UML est le langage standard de la modélisation pour lôorienté objet, il est naturel de lôutiliser comme base pour la définition des éléments de lôapproche Orientée aspect. Il offre également des mécanismes dôextension de ces éléments : les stéréotypes (stereotypes), les valeurs marquées (tagged values) et les contraintes (contraints), permettent lôextension et la spécialisation des classes de concepts et relations standards dôUML. Les stéréotypes permettent dôajouter de nouvelles classes dôéléments de modélisation au métamodèle dôUML,

es, en plus du noyau prédéfini par UML. Les valeurs marquées étendent les attributs des classes dôéléments du métamodele et les contraintes sont des relations sémantiques entre éléments de modélisation qui définissent des conditions que doit vérifier le système. Ces mécanismes ont été exploites dans le cadre de plusieurs travaux, visant a étendre UML pour intégrer les aspects, comme [30], [29], etc.

Un ensemble prédéfini de mécanismes dôextension est appelé profil [16]. Dans notre travail nous allons utiliser ces mécanismes dôextension pour adopter UML à lôapproche orientée aspect.

3. Approche globale

Comme nous avons vu dans le premier chapitre, les travaux récemment proposés [30], [33], [32] et [21] etcé se sont ainsi intéressés à la séparation des préoccupations transversales (aspects) en particulier au niveau de la phase de conception. Toutes les propositions de modélisation par aspects sont dédiées à cette phase et ignorent la phase dôanalyse.

Afin de pallier ce manque, la première étape de notre travail consiste à proposer une approche permettant de décrire les principaux concepts du paradigme aspects. Cette dernière est un profil UML. En intégrant la notion dôaspect non seulement dans les diagrammes de classes mais également dans les diagrammes de cas dôutilisation, notre approche permet de prendre en compte les fonctionnalités de nature transversale dès les premières étapes de la spécification des besoins. Notre approche est de nature itérative et procède donc par raffinements successifs des modèles.

Dans une deuxième étape, nous proposons une méthode de développement de logiciels basée sur la notion dôaspect qui représente les préoccupations de nature transversale et dôune autre part dirigée par les cas dôutilisation dôUML qui explosent bien les besoins des utilisateurs. Cette méthode utilise lôapproche proposée dans la première étape.

4. Approche adoptée : Définition d'un modèle basé sur la notion d'aspect

Comme nous avons mentionné dans la section précédente, notre proposition est un profil UML permettant de décrire les principaux concepts du paradigme aspects. En intégrant la notion dôaspect non seulement dans les diagrammes de classes mais également dans les diagrammes de cas dôutilisation, notre approche permet de prendre en compte les fonctionnalités de nature transversale dès les premières étapes de la spécification des besoins.

otypes, notre profil propose dôenrichir les modèles par des contraintes (dont certaines formulées en OCL) permettant de spécifier formellement les fragments du modèle qui sont pertinents pour la composition des aspects.

Pour illustrer lôutilisation de notre proposition, nous ferons référence à un exemple, qui présente une application simplifiée de simulation dôun système de banque. Cette application permet de simuler un système bancaire dans lequel des usagers peuvent (1) effectuer un virement, (2) retirer de lôargent et (3) consulter leur compte. À ce système de base on souhaite greffer des fonctionnalités transversales que sont le Logging et la vérification du solde. À lôaide de notre approche, nous montrons, dans la section suivante, comment étendre le modèle UML de lôapplication de base afin dôintégrer ces deux nouvelles fonctionnalités.

4.1 Diagramme de cas d'utilisation :

Les aspects peuvent représenter des fonctionnalités auxiliaires, des services secondaires ou des contraintes de qualité et dôutilisation du système. En général, ces traitements ne sont pas représentés dans la vue de cas dôutilisation standard, qui est souvent réservée pour décrire les services de base et les exigences fonctionnelles principales du système. Dans notre approche, nous proposons de considérer les aspects dès la phase dôacquisition des besoins, et de les prendre en charge durant tout le cycle de développement. Pour cela, nous proposons de considérer les aspects comme des cas dôutilisations dôextension et donc introduire un nouveau élément appelé **Aspect use-case .**

4.1.1 Aspect use-case :

- **Définition**

Lôaspect est lôunité modulaire dôencapsulation dôune préoccupation transversale (crosscutting concern). Les aspects peuvent représenter des fonctionnalités auxiliaires, des services secondaires (Ex : sécurité, performance, Logging, é é).

- **Notation graphique :**

Un cas dôutilisation aspect se représente de la même façon dôun cas dôutilisation UML, par une ellipse contenant le nom de lôaspect (nom ou verbe à lôinfinitif), et optionnellement, au-dessus du nom dôaspect, un stéréotype ***« aspect »***. La figure suivante montre la représentation dôun cas dôutilisation aspect.

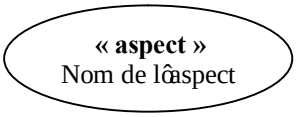


FIG.3.1 ***Représentation d'un aspect use -case***

4.1.2 Ajout des aspects au niveau du diagramme de cas d'utilisation

Comme mentionné au-dessus les aspects sont considérés comme des dôutilisation dôextension par rapport au cas dôutilisation de base. Les points de coupure sont alors assimilés aux points dôextension, i.e. un ensemble de locations où lôon peut tisser les cas dôutilisation qui représentent un aspect. Pour modéliser le comportement transverse de ces aspects et leur insertion dans les cas dôutilisation de base, la relation ***« extend »*** de UML est utilisée. Les cas dôutilisation de base seront donc modifiés implicitement aux *points d'extension* indiqués.

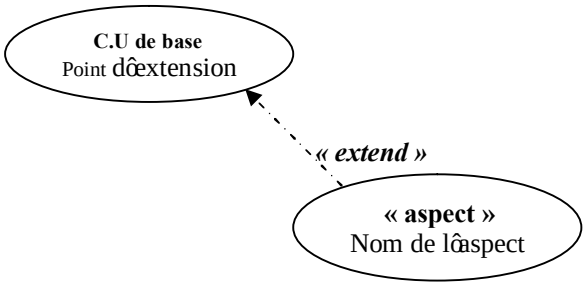
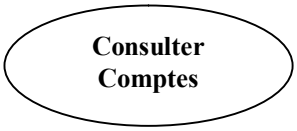


FIG.3.2 ***Ajout des aspects au Diagramme de cas d'utilisation***

Dans notre exemple de lôapplication du système bancaire, nous avons les fonctionnalités de base *retirer de l'argent*, *effectuer un virement* et *consulter compte*, représentées respectivement par les cas dôutilisation de base portant les mêmes noms. Lôintégration à lôapplication « système bancaire » des deux fonctionnalités auxiliaires *Logging et vérifier solde*, se fait grâce à la notion dôaspects. On introduit deux aspects *Logging et vérifier solde*.

Par conséquent, ces deux aspects entrecoupent la fonctionnalité de base *effectuer un virement*. Dans la vue de cas dôutilisation proposée à la figure 5, les deux aspects *Logging et vérifier solde*, sont considérés comme des cas dôutilisation dôextension. Pour modéliser le comportement transverse de ces deux aspects et leur insertion dans le cas dôutilisation de base *effectuer un virement*, la relation « extend » de UML est utilisée. Le cas dôutilisation de base sera donc modifié implicitement au point dôextension indiquée « vérifier_solde » nommé *EpCompte*.



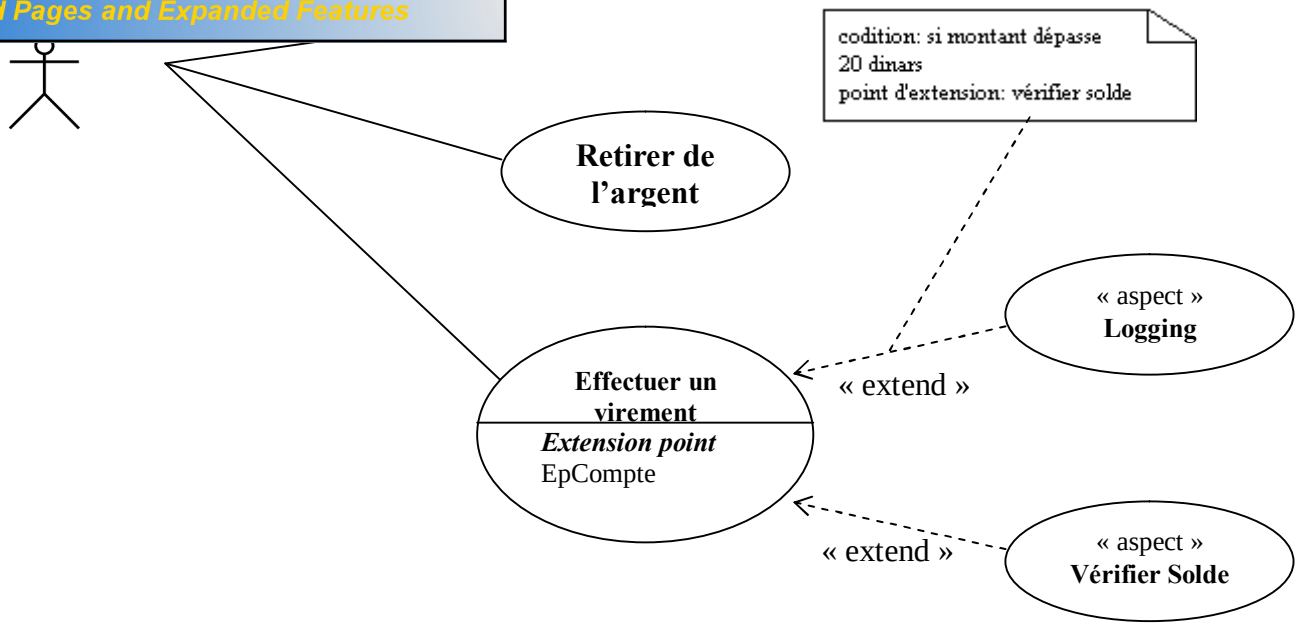


FIG.3.3 ***Logging et Vérifier Solde exprimés comme cas d'utilisation d'extension***

4.2Diagramme de classes :

4.2.1 classe « Aspect » :

- **Sémantique :**

Lôaspect est une classe qui représente une abstraction dôune préoccupation transversale (crosscutting concern) et est utilisé pour modéliser cette dernière.

- **Notation graphique :**

Comme les classes, les aspects sont représentés par des rectangles contenant des compartiments**.** Le premier compartiment symbolise lôaspect, il contient son nom et le stéréotype **« aspect ».** Le deuxième contient les attributs propres à lôaspect et troisième contient les méthodes relatives à lôaspect ainsi que ses points de coupure et leurs advices.

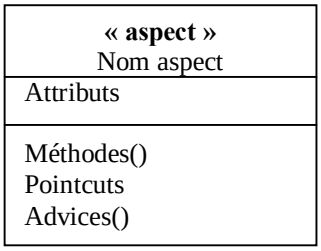


FIG.3.4 ***Notation graphique d'un aspect et ses éléments***

4.2.2 classe « PointCut » :

de points de jointure.

- **Notation graphique :**

Un point de coupure est modélisé par une interface, cette dernière est représentée par un rectangle contenant deux compartiments : le premier symbolise le point de coupure ; il contient son nom et optionnellement au-dessus un stéréotype **« PointCut ».** Le second compartiment contient des opérations abstraites qui doivent être exécutées quand un des points de jointure liés au point de coupure en question est atteint.

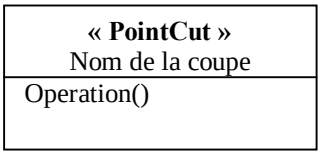


FIG.3.5 Notation graphique d'un point de coupure

4.2.3 Relation d'association : « Crosscut »

- **Sémantique :**

La relation stéréotypée « Crosscut » est utilisée pour modéliser la relation d'entrecroisement quand le code de l'aspect entrecoupe le corps du composant fonctionnel. C'est une relation de dépendance entre une interface « PointCut » et ses points de jointures correspondants aux méthodes des classes entrecroisées.

- **Notation graphique :**

Une relation « Crosscut » est modélisée par un trait discontinu terminé par une flèche triangulaire et portant le stéréotype **« Crosscut ».**

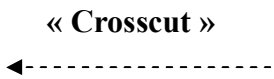


FIG.3.6 Notation graphique de relation d'entrecroisement

Considérons les cas d'utilisation *Logging et vérifier solde* de notre exemple. La figure 9 montre comment les exigences de ces fonctionnalités sont intégrées dans le diagramme de classes UML. Les comportements des cas d'utilisation *Logging et vérifier solde* sont décrits par des classes nommées respectivement *Logging et vérifier solde* portant le stéréotype « aspect ». Quant aux point d'extension *EpCompte* , il correspond à un point de coupure et est donc modélisés par une interface portant le stéréotype « Pointcut ». Un point de coupure est lié à un ensemble de points de jointure. Une relation de dépendance « crosscut » relie donc

ointure. Lôinterface *EpCompte* a une relation « crosscut » pointant respectivement vers les points de jointure correspondant à la méthode *créditer ()* et à la méthode *débiter ()* de la classe *Compte.*

Lôinterface *EpCompte* contient une opération abstraite nommée respectivement *opCompte* qui doit être exécutée quand un de ses points de jointure est atteint. Lôaspect *Logging* implémente lôinterface *EpCompte* et propose donc une méthode, appelée advice, qui sera exécutée aux points de jointure spécifiés. Une relation de réalisation lie la classe « aspect » à lôinterface « pointcut ». À remarquer quôun advice est annoté avec lôun des stéréotypes « before », « after » ou « around » selon quôil est exécuté respectivement avant, après ou autour des points de jointure référencés par le point de coupure.

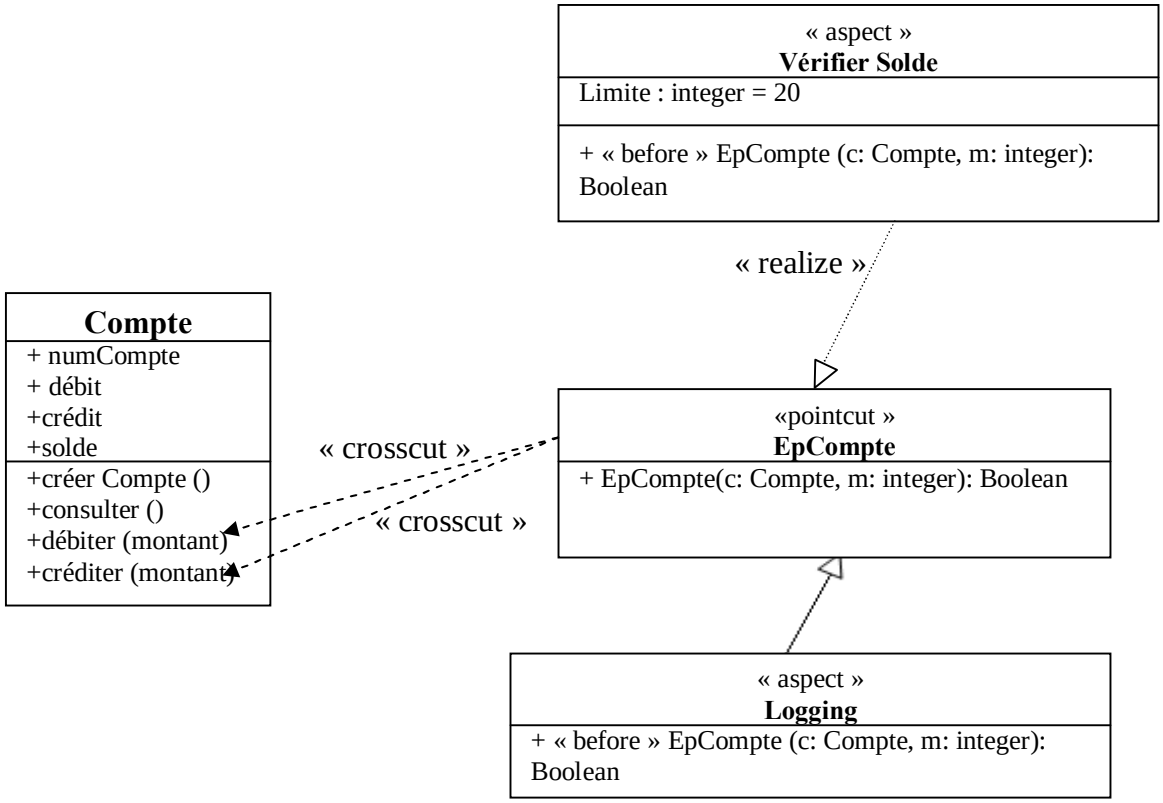


FIG.3.7 Diagramme de classes de l'application « système bancaire »

4.3Contraintes de composition :

Nous devons enrichir le diagramme de classes avec des contraintes formelles dont certaines sont exprimées dans le langage OCL. Notre modèle requiert les sortes de contraintes suivantes :

st possible que plusieurs aspects sôentrecoupent à un même point de jointure et que tous proposent un advice de la même sorte (*before* ou *after*). Sur un diagramme de classes on peut identifier les aspects conflictuels en repérant les advices de la même sorte qui implémentent une même interface « Pointcut ».Il est toutefois possible de résoudre les conflits en définissant une relation de préférence entre les aspects. Pour ce faire, il suffit dôajouter au diagramme de classes une contrainte globale de la forme *aspect1*< *aspect2* où < est une relation dôordre partiel. Dans le cas de lôapplication « système bancaire », puisque la vérification du solde doit être exécutée avant lôenregistrement de toute transaction de retrait ou de virement (le logging) ; cela est exprimée par la contrainte *vérifier solde* < *logging* .

- une classe ne peut jamais hériter dôun aspect : cela est exprimé en OCL par :

context Generalization inv :
self.child.oclIsKindOf(Class) implies not(self.parent.oclIsKindOf(Aspect))

- « realize »: spécifie une relation de réalisation qui lie un aspect à son interface « pointcut »
- « crosscut » : spécifie la relation dôentrecroisement qui lie une interface « pointcut » à ses points de jointure.
- **Spécification des points de jointure :**

Nous devons fournir la spécification des méthodes constituant des points de jointure. Définie indépendamment du contexte dôexécution de la méthode, cette spécification est exprimée en OCL par des pré et post conditions et est notée de la façon suivante :

context *nom_classe* : :*nom_méthode()*
pre : condition1,…
post : condition2,…

Le nom du contexte est celui de la méthode (préfixé par le nom de la classe à laquelle elle appartient), constituant le point de jointure. Les conditions sont des expressions booléennes respectant la syntaxe et la sémantique dôOCL. Une pré-condition exprime une propriété nécessaire à lôexécution de la méthode, alors quôune post-condition exprime une propriété assurée suite à lôexécution de la méthode. Pour lôapplication « système bancaire », la spécification du point de jointure *Compte::créditer()* est comme suit :

context *Compte* : :*créditer()*
pre : état = 1
post : état= 2

de points de jointure et de préférence peuvent être spécifiées directement sur le diagramme de classes au moyen dôune note et quôon rattache à lôélément contraint.

5. Processus de développement orienté aspect et dirigé par les cas d'utilisation

Après avoir défini notre modèle dans la première section de ce chapitre, nous proposons dans cette partie une méthode de développement basée sur la notion dôaspect qui représentent les besoins non fonctionnels et orientée utilisateur. Cette méthode utilise les principes définis précédemment dans lôapproche proposée dans la section 4.

Lôarchitecture se définit en considérant dôune part une spécification des besoins non fonctionnels et d'autre part les besoins fonctionnels du système définis par des cas d'utilisation.

En général le développement de logiciels a trois étapes essentielles, montrées sur la figure3.8.

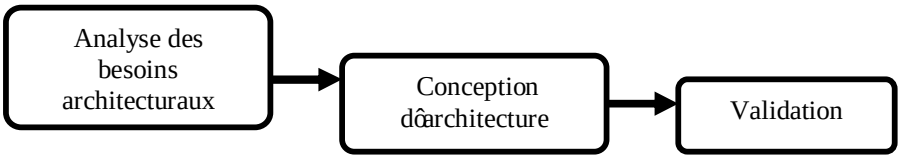


Fig.3.8 - Trois phases essentielles de développement des logiciels.

Dans ce qui suit nous présentons d'abord la méthode, ensuite nous appliquons cette méthode pour le développement d'un système de banque.

La méthode proposée ici consiste en deux phases : la phase analyse et la phase conception. La figure 3.9 montre cette méthode de développement. Nous présentons ici ses différentes étapes:

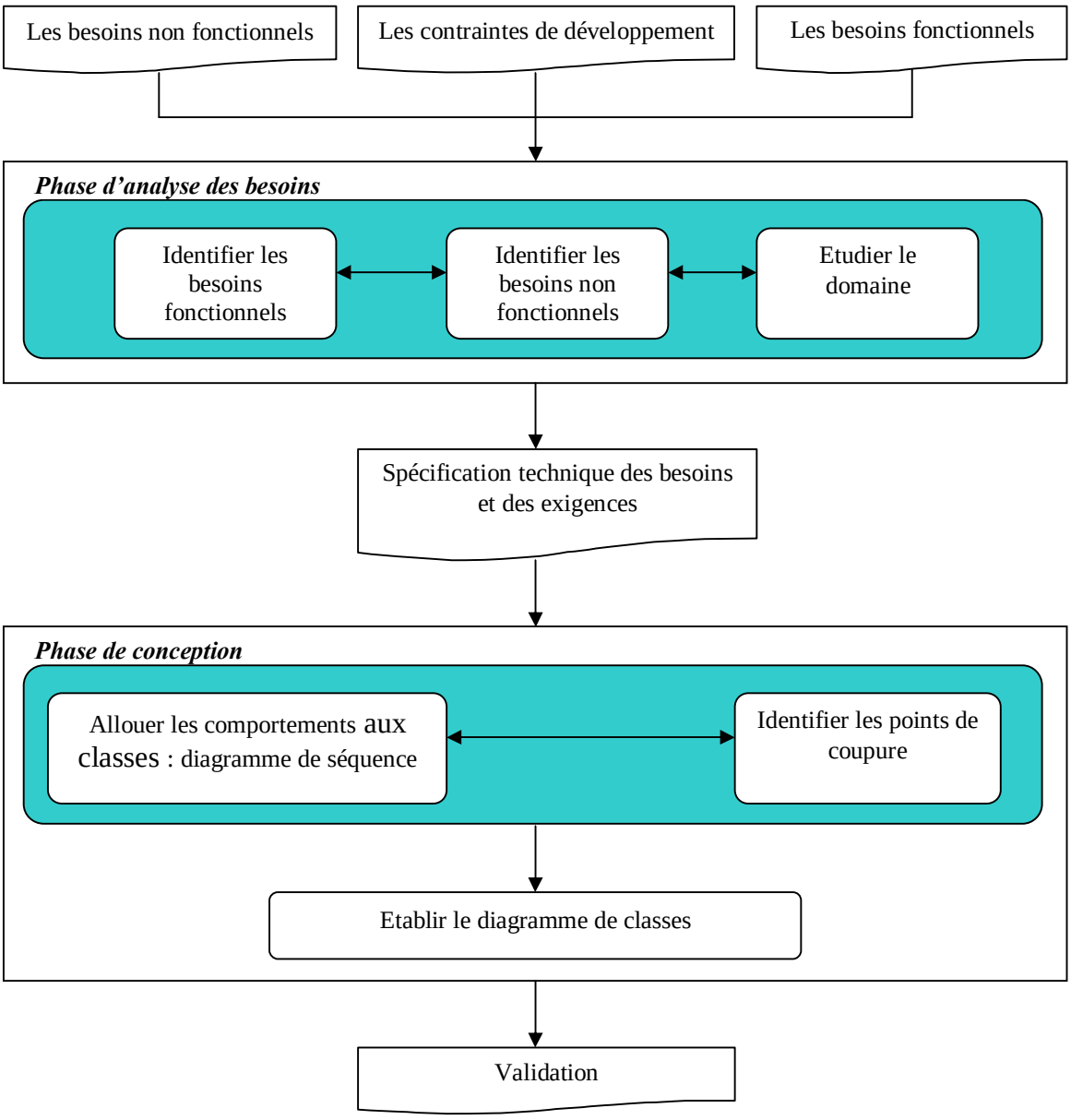


FIG. 3.9 Processus de développement

5.1 Obtention du Cahier de Charges

Avant de commencer l'analyse nous devons fournir le cahier des charges. La phase d'analyse des besoins commencera à partir des informations fournies par le client dans le cahier des charges, en langage naturel. Ce cahier des charges doit comporter les informations suivantes :

- **Les besoins fonctionnels :** représentant les fonctionnalités attendues du système.

ls : représentant les propriétés non fonctionnelles attendues du système par les clients.

- **Les contraintes du développement :** représentant les contraintes concernant le système, aussi bien concernant son fonctionnement comme le système d'exploitation, que le développement comme les contraintes de coût ou de temps, etc.

5.2 Phase d'analyse des besoins

La phase d'analyse des besoins se base sur le cahier des charges. Cette phase comporte trois activités que les architectes peuvent traiter parallèlement ou dans un ordre choisi.
Les flèches bidirectionnelles entre les activités montre la dépendance entre elles. Les activités de la phase d'analyse peuvent s'effectuer en parallèle.

- **Identifier les besoins fonctionnels :**

En utilisant les différents diagrammes tel que le diagramme de cas d'utilisation, il est possible de vérifier la complétude et la cohérence des fonctionnalités requises du système. Les besoins fonctionnels seront réécrits en fonction des exigences des utilisateurs. Au cas d'incohérence, le cahier des charges doit être modifié avec le client.
Les cas d'utilisation sont utilisés tout au long du projet. Dans un premier temps, on les crée pour identifier et modéliser les besoins des utilisateurs. Ces besoins sont déterminés à partir des informations recueillies lors des rencontres entre informaticiens et utilisateurs. Il faut impérativement prendre en compte toute considération de réalisation lors de cette étape. Durant cette étape, il faut :

- Identifier les acteurs.
- Recenser les cas d'utilisation.

- **Identifier les besoins non fonctionnels :**

En ce qui concerne les besoins non fonctionnels, ils sont considérés comme des aspects, ces derniers peuvent représenter des fonctionnalités auxiliaires, des services secondaires ou des contraintes de qualité et d'utilisation du système. En général, ces traitements ne sont pas représentés dans la vue de cas d'utilisation standard, qui est souvent réservée pour décrire les services de base et les exigences fonctionnelles principales du système. Dans notre approche, nous proposons de considérer les aspects dès la phase d'acquisition des besoins, et de les prendre en charge durant tout le cycle de développement. Pour cela, nous proposons de considérer les aspects comme des *cas d'utilisations d'extension* par

ication (ou de base). Les *points de coupure* sont alors assimilés aux *points d'extension*, (i.e. un ensemble de locations où lôon peut tisser les cas dôutilisation qui représentent un aspect). Pour modéliser le comportement transverse de ces aspects et leur insertion dans les cas dôutilisation de base, la relation « extend » de UML est utilisée. Les cas dôutilisation de base seront donc modifiés implicitement au point dôextension indiqué.

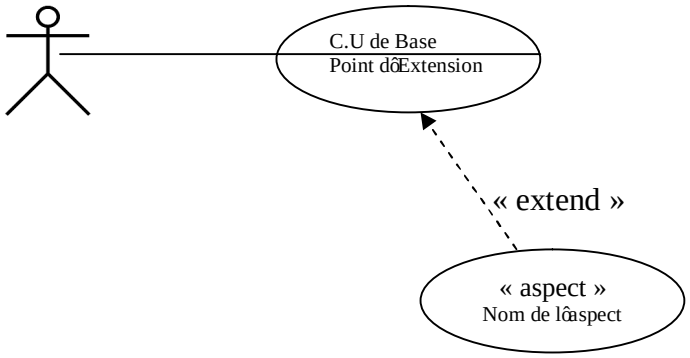


FIG.3.10 ***Diagramme de cas d'utilisation avec l'ajout des aspects***

- **Etudier le domaine : identification des classes participantes**

  La première étape quand on analyse un cas dôutilisation est lôidentification des classes participantes dans la réalisation de ce cas.

  Pour chaque cas dôutilisation, nous identifions les classes métiers qui participent à ce cas dôutilisation. En fait une classe métier correspond à un concept métier. Par exemple, pour le cas (« Effectuer un virement»), nous identifions facilement les concepts métier «Compte» et « Crédit ». Nous allons modéliser ces concepts sous forme de diagrammes de classes contenant uniquement des attributs et des associations. Pour le moment nous ne faisons pas ressortir les méthodes.

Les résultat de ces étapes est : le cahier des *Spécifications Techniques des Besoins et des Exigences.*

Avant de passer à lôétape suivante, nous faisons un point sur les sorties des étapes précédentes:

- Le document *"Spécification technique des besoins et des exigences"* explicite les besoins fonctionnels du système, réécrits par l'analyste et vérifiés et validés par le client. Ainsi que les besoins non fonctionnels de chaque fonctionnalité.

Nous utilisons les résultats de la phase précédente, d'analyse des besoins, comme données de cette phase.

En fait, lôarchitecture du système commence à prendre forme dès la spécification des besoins. Puis, elle évolue à mesure que lôon avance dans le processus. Toujours pour chacun des cas dôutilisation, on identifie les classes dôanalyse qui vont participer à la réalisation des cas dôutilisation. Pour cela, nous utilisons la décomposition préconisée par Ivar Jacobson [34], qui propose que la réalisation dôun cas dôutilisation soit faite à travers la collaboration de trois types dôobjet :

ÅLes « dialogues (ou interfaces)» qui représentent les moyens dôinteraction avec le système.

ÅLes « contrôles (ou traitement fonctionnel)» qui contiennent la logique applicative.

ÅLes « entités (ou données)» qui spécifient la logique métier.

- **Allouer les comportements aux cas d'utilisation : spécifier chaque cas d'utilisation**

Introduire les fonctionnalités principales pour chaque cas d'utilisation en utilisant différents diagrammes, par exemple les diagrammes de séquence ou les diagrammes de communication.

Une fois les classes participantes sont identifiées, nous procédons à lôétape suivante qui permet de décrire comment les objets collaborent afin de réaliser les cas dôutilisation. Cela est achevé en décrivant comment les instances de classes interagissent entre elles. Pour montrer ces interactions, nous allons utiliser le diagramme de séquence qui nous permettra dôajouter des attributs et des opérations dans les nouvelles classes de conception, ainsi que des associations entre elles. Dans cette étape, nous allons remplacer le système vu comme une boite noire par un ensemble dôobjets communicants en utilisant les trois types de classes, à savoir les « dialogues », les « contrôles » et les « entités »

- **Identifier les points de coupure :**

Nous avons vu auparavant que les cas dôutilisation de base seront étendus par les fonctionnalités transversales à un point précis appelé *Point d'extension*, ce dernier sera transformé en un *point de coupure* au niveau des cas dôutilisation dôextension représentant les aspects et est donc modélisé par une interface portant le stéréotype *« PointCut ».*

e conception:

En partant des interactions -montrées au niveau du diagramme de séquence- entre les classes participantes à la réalisation du cas dôutilisation, nous allons attribuer les responsabilités et les relations requises à ces dernières dans le contexte du cas dôutilisation.

Considérons les cas dôutilisation *d'extension*. Les comportements des cas dôutilisation dôextension sont décrits par des classes portant le stéréotype « Aspect ». Quant aux points dôextension, ils correspondent à des points de coupure et sont donc modélisés par des interfaces portant le stéréotype *« Pointcut ».* Un point de coupure est lié à un ensemble de points de jointure.

Une relation de dépendance *« crosscuts »* relie donc chaque interface *« Pointcut »* à ses points de jointure.

Les interfaces contiennent chacune une opération abstraite qui doit être exécutée quand un de leurs points de jointure est atteint. Lôaspect implémente donc les interfaces et propose donc des méthodes, appelées advices, qui seront exécutées aux points de jointure spécifiés. À remarquer quôun advice est annoté avec lôun des stéréotypes *«before», « after » ou « around »* selon quôil est exécuté respectivement avant, après ou autour des points de jointure référencés par le point de coupure.

6. Conclusion

Dans ce chapitre, nous avons proposé une méthode de développement basée sur la notion dôaspect et orientée utilisateur (centrée sur les cas dôutilisation UML). Dans un premier temps notre travail expose une approche pour la modélisation des systèmes par aspects. Nous avons dôabord présenté un profil UML comme notation de modélisation intégrant les constructions du paradigme aspect. Ce profil permet, entre autres, de mettre en évidence les contraintes de précédence entre aspects et les propriétés caractérisant le contexte de leur composition. Ensuite nous avons donné le processus de développement de la méthode proposée en se basant sur le profil proposé. Ainsi, c'est dès lôacquisition des besoins qu'il est possible de s'assurer de la prise en compte des différentes fonctionnalités demandées à un système.

Notre expérience montre que les propriétés non fonctionnelles ne peuvent pas être étudiées indépendamment des besoins fonctionnels et que leur prise en compte dès la phase dôanalyse facilite leur modélisation au niveau de la conception et de lôimplémentation (grâce à la notion

l en compte d'abord les besoins fonctionnels, ensuite pour chaque fonctionnalité elle vérifie ses propriétés non fonctionnelles.

Afin de valider notre approche, le chapitre suivant présente une étude de cas permettant dôappliquer notre méthode pour le développement dôun système de gestion dôun compte bancaire.

Etude de cas : Un système de gestion d'un compte bancaire

4

Ce chapitre couvre

- *Introduction*
- *Cahier de charges*
- *Application de la méthode*
- *Implémentation de quelques aspects et exemples*
- *Evaluation du processus*
- *Conclusion*

Etude de cas : Un Système de Gestion dôun Compte Bancaire

1. Introduction

Afin dôillustrer et de valider la démarche de conception dôarchitecture exposée dans les parties précédentes, nous allons lôappliquer à une étude de cas : une unité dôun système de Banques.

La modélisation dôun Système dôInformations dôune Banque est un travail fastidieux compte tenu de la complexité du système et du nombre important dôacteurs quôil peut impliquer.

Toutefois, pour notre projet, nous avons du procéder à quelques simplifications, faute de temps, afin de pouvoir mettre en évidence lôapport de la modélisation UML pour la mise en place dôArchitectures Orientées Aspect (AOA en anglais pour Aspect Oriented Architecture).

Dans ce qui suit, nous allons suivre la démarche proposée durant le chapitre 3 afin de modéliser le Système dôInformations (SI) dôune Banque X, selon un cahier de charges bien défini, le but étant dôimplémenter ultérieurement un logiciel qui répond aux besoins soulevés par la présente étude.

Nous définissons le cahier des charges et nous appliquons notre méthode de développement pour concevoir un tel système.

2. Cahier des charges du système « Banque »

2.1 Architecture fonctionnelle du SI :

Il sôagit dôimplémenter un mini système bancaire. Ce système bancaire se compose dôune banque qui possède un certain nombre de clients. Chaque client possède un ou plusieurs comptes dans cette banque. La figure 4.1 montre lôarchitecture fonctionnelle du système « Banque ».

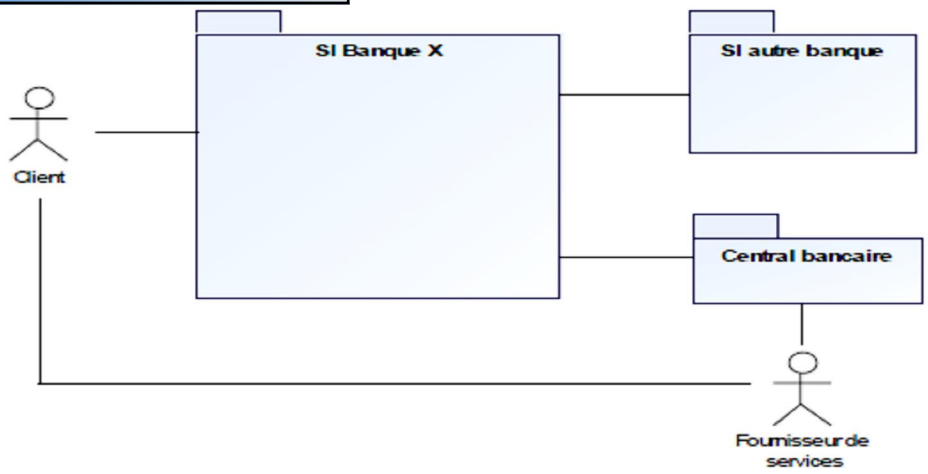


FIG.4.1 Diagramme de contexte (boîte noire)

2. 2 Elicitation des différents flux :

La banque possède un GAB (pour guichet automatique de banque) auprès duquel il est possible de réaliser les actions suivantes :

A. Différents flux :

- Création dûun compte,
- Clôture dûun compte,
- Opération dûouverture de crédit,
- Retrait dûargent (à lûagence de la Banque X, au GAB de la Banque X, au GAB dûune autre banque),
- Dépôt dûargent,
- Demande de virement (de compte à compte : à lûagence ou au GAB de la Banque X, de compte à compte : banque X - autre banque au guichet de la Banque X),
- Consultation des comptes (au guichet, au GAB de la Banque X ou sur Internet),
- Echange de devise,
- Achat et paiement par Internet,
- Demande de chéquiers (à lûagence ou par Internet).
- Demande dûinformation.

es types de données :

- **Création d'un compte :**

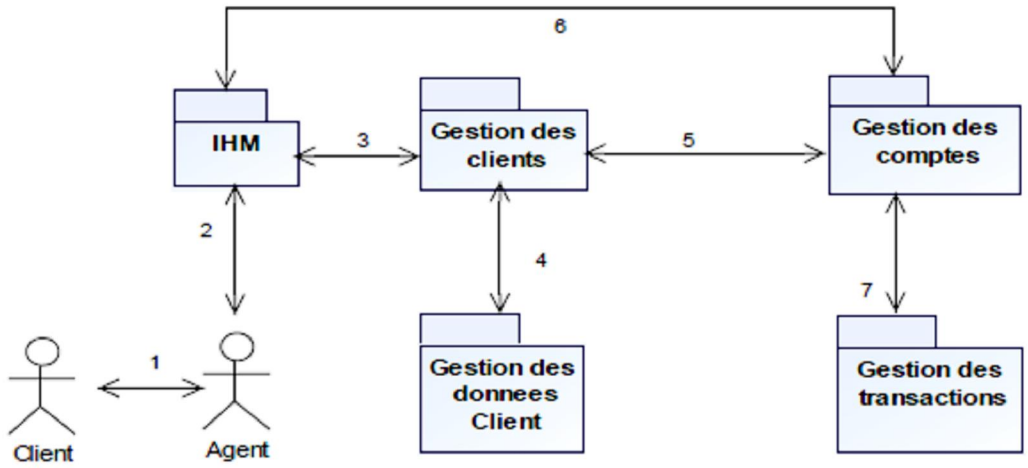


FIG.4.2 Flux du cas « Création d'un compte »

Flux	Description
1	Lorsquôun client souhaite ouvrir un compte, il se présente dans lôagence de sa Banque X et rencontre un agent bancaire.
2	Lôagent, par la demande de création dôun compte, va ouvrir un compte au nom du client. Il accède via une application de gestion.
3-4-5-6	Sôil sôagit dôun nouveau client, lôagent va créer son fichier client et va créer le compte. Sôil est déjà client de la banque, lôagent accède à son fichier client et visualise les comptes existants.
7	Une fois le compte créé, lôagent demande ensuite au client le montant initial quôil souhaite déposer.

- **Retrait d'argent**

Il y a 3 possibilités de retrait :

- à lôagence de la Banque X,
- au GAB de la Banque X,
- au GAB dôune autre banque.

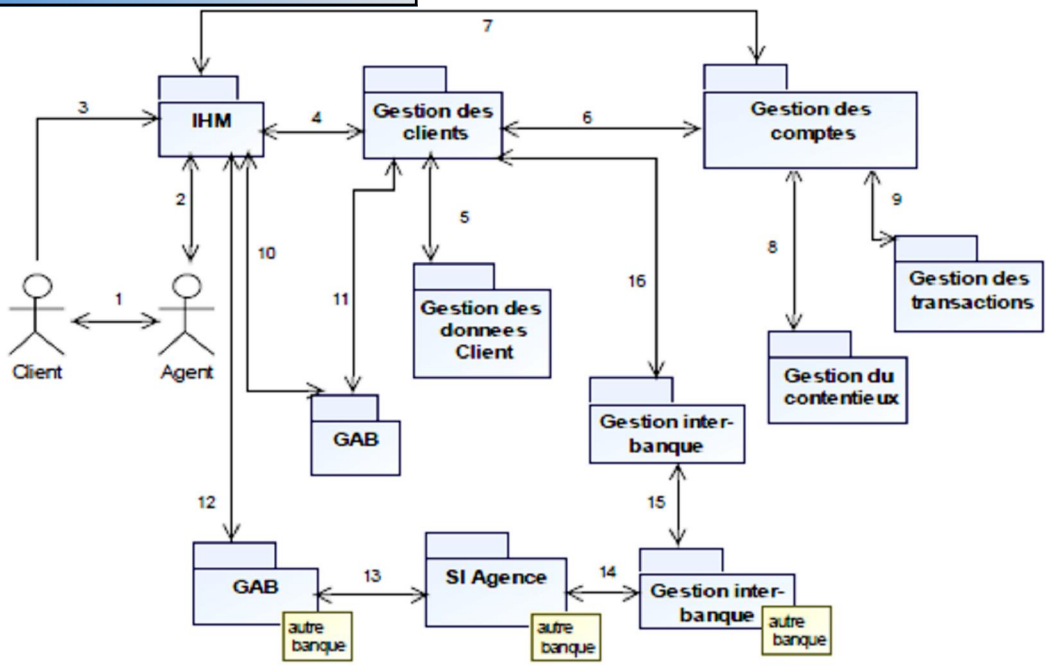


FIG.4.3 Flux du cas « **Retrait d'argent** »

❖ **A l'agence de la Banque X :**

Flux	Description
1-2	Le client se présente directement à lôagence pour retirer de lôargent.
4-5-6-7	Lôagent bancaire consulte le fichier du client en lui demandant sur quel compte le client souhaite effectuer le retrait.
8-9	Lôagent vérifie si le client est accordé à effectuer des retraits (somme disponible, le client nôest pas en découverté). Dans le cas où le retrait est accordé, lôagent effectue les opérations de retrait. Le client récupère lôargent et le reçu.

❖ **Au GAB de la Banque X :**

Flux	Description
3-10	Le client souhaite effectuer le retrait au GAB de la Banque X.
11-5-6-7	Le client accède au menu principal (opérations au préalable : insertion de la carte, saisie du code et validation + si bonne authentification, choix de lôopération de retrait sur le compte souhaité en précisant le montant à retirer).
8-9	Si le client est accordé à effectuer des retraits (somme disponible, le client nôest pas en découverté), le GAB lui délivre le montant correspondant.

que :

Flux	Description
3-12	Le client souhaite effectuer le retrait au GAB dôune autre banque.
13-14-15-16-5-6	Le GAB interroge la Banque X et le compte du client. Le client accède au menu principal (opérations au préalable : insertion de la carte, saisie du code et validation + si bonne authentification, choix de lôopération de retrait sur le compte souhaité en précisant le montant à retirer).
8-9	Si le client est accordé à effectuer des retraits (somme disponible, le client nôest pas en découverté), le GAB lui délivre le montant correspondant.

- **Dépôt d'argent**

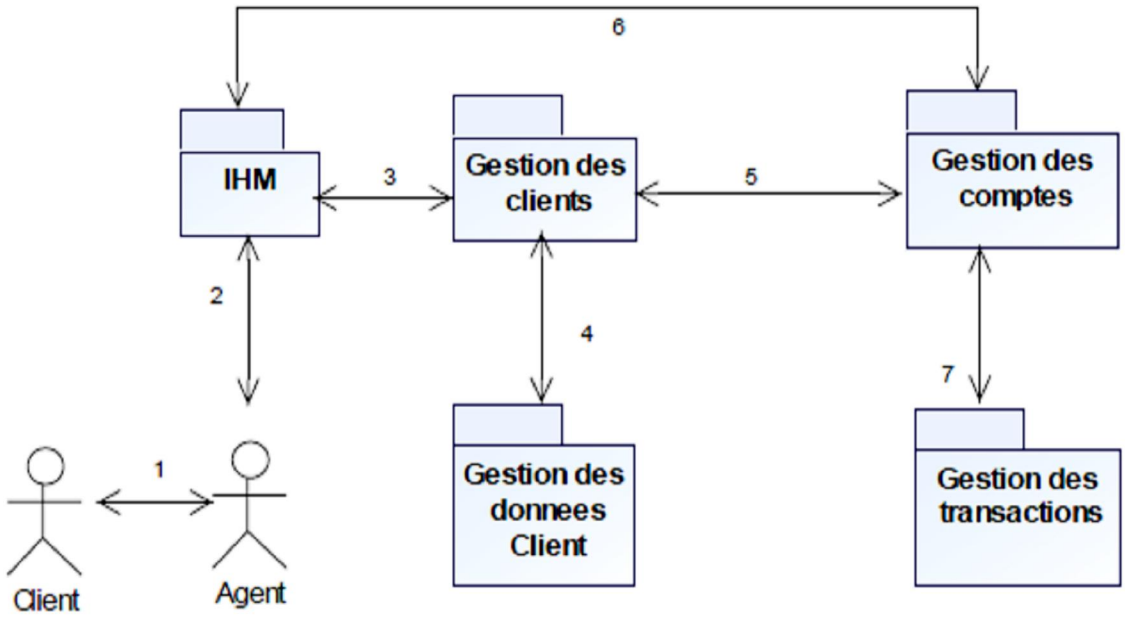


FIG.4.4 Flux du cas «**Dépôt d'argent** »

Flux	Description
1-2	Lôagent souhaite effectuer un dépôt dôargent à la Banque X. Il rencontre lôagent bancaire.
3-4-5-6	Lôagent accède au fichier du client.
7	Lôagent accède au compte du client et lui demande la somme quôil souhaite déposer.

Le client peut effectuer un virement, soit :

- entre comptes de la Banque X : à lôagence ou au GAB de la Banque X.
- dôun compte de la Banque X vers un compte dôune autre banque (au guichet de la Banque X).

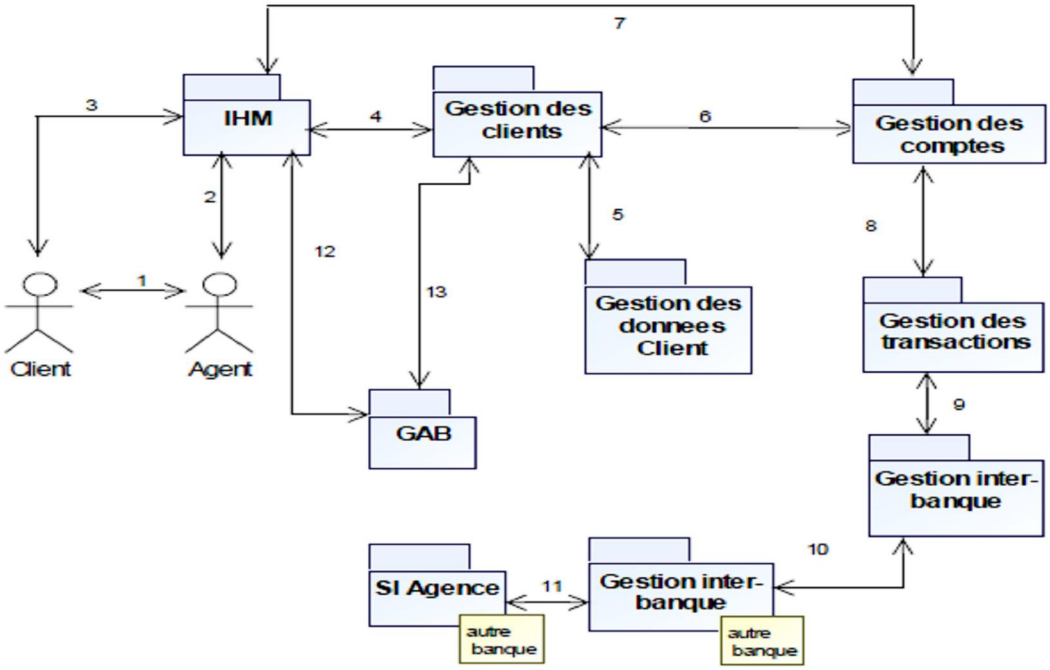


FIG.4.5 Flux du cas «**Demande de virement** »

Flux	Description
1-2	Le client se présente à lôagence de la Banque X pour effectuer un virement. Il rencontre donc lôagent bancaire.
3-12-13	Il peut également le faire via le site sécurisé de la Banque X ou au GAB de la Banque X.
4-5-6-7	Accès au fichier du client et à ses comptes.
8	Si le client fait le virement à lôagence, lôagent lui demande quel est le compte à débiter et celui à créditer, ainsi que le montant du virement. Si côest au GAB ou sur Internet, le client précise les informations lui-même.
9-10-11	Un virement peut être effectué dôun compte de la Banque X vers un compte dôune autre banque. Dans ce cas, lôagent lui demande les informations nécessaires au virement (n° des comptes, montanté).

Cela peut se faire soit :

- au guichet,
- au GAB de la Banque X,
- ou sur Internet.

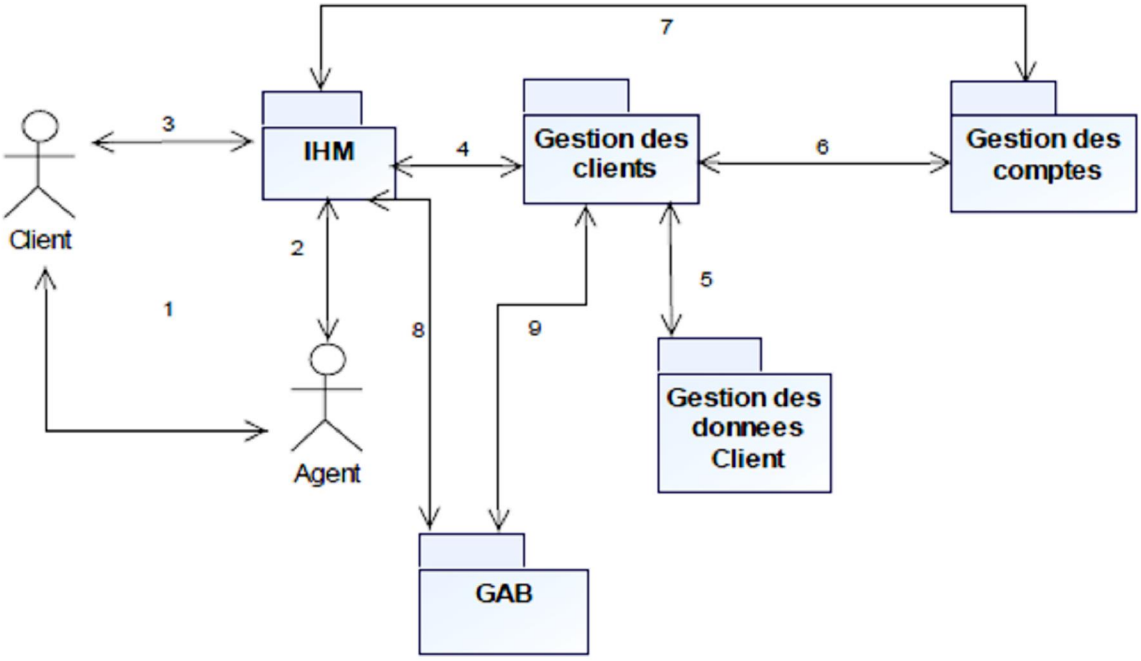


FIG.4.6 Flux du cas «Consultation des comptes»

Flux	Description
1-2	Le client se présente à lôagence de la Banque X pour consulter ses comptes. Il rencontre donc lôagent bancaire.
3-8	Il peut également consulter ses comptes au GAB ou par Internet sur le site sécurisé de la Banque X.
4-5-6-7	Si le client consulte ses comptes dans lôagence, lôagent accède directement à son fichier client et à ses comptes.
9	Au GAB, la consultation des comptes par le client est une opération. Il accède donc à ses comptes.

2.3 Contraintes de développement et choix techniques :

- Installation de lôenvironnement Eclipse : côest un environnement de développement intégré (Integrated Development Environment) dont le but de fournir une plateī forme modulaire pour permettre de réaliser des développements informatiques.

le développement Web est PHP/MySQL et cêest la solution technique que nous allons adopter pour développer la partie Web de notre projet.

3. Application de la méthode au système de gestion d'un compte bancaire

Nous allons appliquer notre méthode vue dans le chapitre 3 pour développer un système de gestion dôun compte bancaire.

3.1 Phase d'analyse des besoins :

La phase d'analyse des besoins a trois activités.

1. Identifier les besoins fonctionnels :

En effet, dans lôexemple, lôacteur le plus important pour un système de gestion de compte bancaire est bien entendu le ***client***. Pour ce problème les cas dôutilisation suivants sont identifiés :

- Consulter un Compte
- Retirer de lôargent
- Faire un virement

Le diagramme suivant montre les cas d'utilisation de ce système.

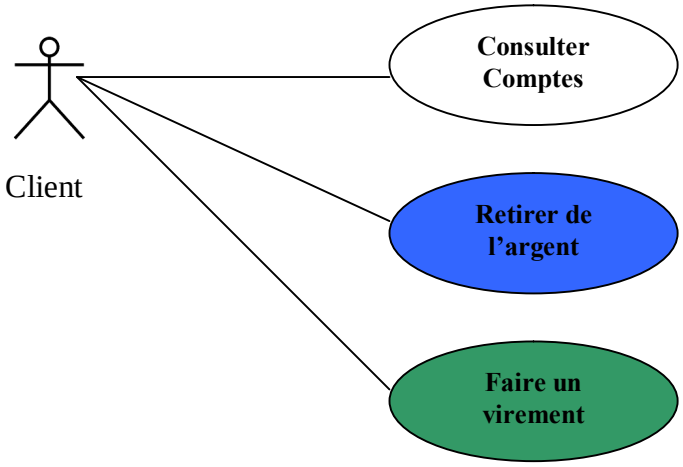


FIG.4.7 Cas d'utilisation du système de gestion d'un compte bancaire.

Nous nous intéresserons plus particulièrement dans la suite de cet article au cas dôutilisation « **Retirer de l'argent** » et « **Faire un virement** », comme illustré sur le diagramme de cas dôutilisation simplifié ci-avant.

ription textuelle détaillée des cas dôutilisation qui nous concernent (le style utilisé est celui préconisé par A. Cockburn [23] dans son récent ouvrage de référence : « Rédiger des cas dôutilisation efficaces », Eyrolles, 2001).

CU « Retirer Argent au GAB » (de la Banque X) :

- **Patron textuel :**

Nom :

Retirer Argent au GAB (de la Banque X).

Acteurs :

Le Client, Banque X.

Résumé :

Un Client de la Banque X souhaiterait retirer une somme dôargent du GAB dôune agence de la

Banque X ou dôune autre banque.

Pré Conditions :

Le solde du client doit être supérieur à la somme à retirer.

Le Client doit avoir sa carte bleue sur lui.

Le GAB fonctionne.

Le GAB doit contenir les coupures de billets nécessaires.

Le GAB doit contenir du papier pour lôimpression du reçu.

Cas nominal :

Le Client introduit sa carte, elle est identifiée par le GAB, le GAB demande le type de transaction et sôil sôagit dôun retrait le client tape son code puis la somme demandée. Le GAB fait appel au Back Office de la banque pour mettre à jour le solde du compte. Lôargent est donné, avec la carte et le ticket.

Cas dérivé :

La carte nôest pas identifiée.

Le code est erroné.

La carte est avalée.

Il nôy a pas de papier dans lôimprimante.

Le solde du compte est insuffisant.

Post Condition :

Le Client est satisfait et il sôen va.

UC « Faire virement » (à lôagence de la Banque X) :

- **Patron textuel :**

Nom :

Faire virement (à lôagence de la Banque X).

Acteurs :

Le guichetier, le client et la Banque centrale.

Résumé :

Un Client de la Banque X souhaiterait effectuer le virement dôun certain montant auprès de son agence.

Pré Condition :

Le solde du Client doit être supérieur à la somme à virer.

Le Client doit disposer de son RIB et de sa pièce dôidentification.

Le Client doit disposer des coordonnées bancaires du compte dans lequel doit aboutir le virement.

Le guichetier doit être libre.

Cas nominal :

Le guichetier récupère toutes les informations nécessaires pour le virement auprès du client, il procède à la vérification des données et à lôenregistrement de lôopération dans la base de données et dans le compte du client. Le guichetier soumet cette opération à la Banque centrale qui doit donc autoriser cette transaction. Le virement a donc été déclenché.

Cas dérivé :

Les coordonnées bancaires soumises au guichetier sont fausses

Le Client ne dispose pas sur son compte dôune somme suffisante pour effectuer le virement.

Le guichetier nôest pas libre.

Le Client peut souhaiter faire dôautres opérations comme un retrait ou une consultation du compte.

Post Condition :

Le guichetier remet un bordereau de confirmation au Client.

2. Identifier les besoins non fonctionnels

Nous définissons les besoins non fonctionnels reliés à chacun des besoins fonctionnels.

Toute transaction de retrait ou de virement doit être enregistrée afin dôêtre sauvegardée dans un fichier persistant sur un serveur central.

Vérifier le solde :

Ne pas effectuer une transaction qui excède 20 dinars. Cette transaction requiert certaine autorisation.

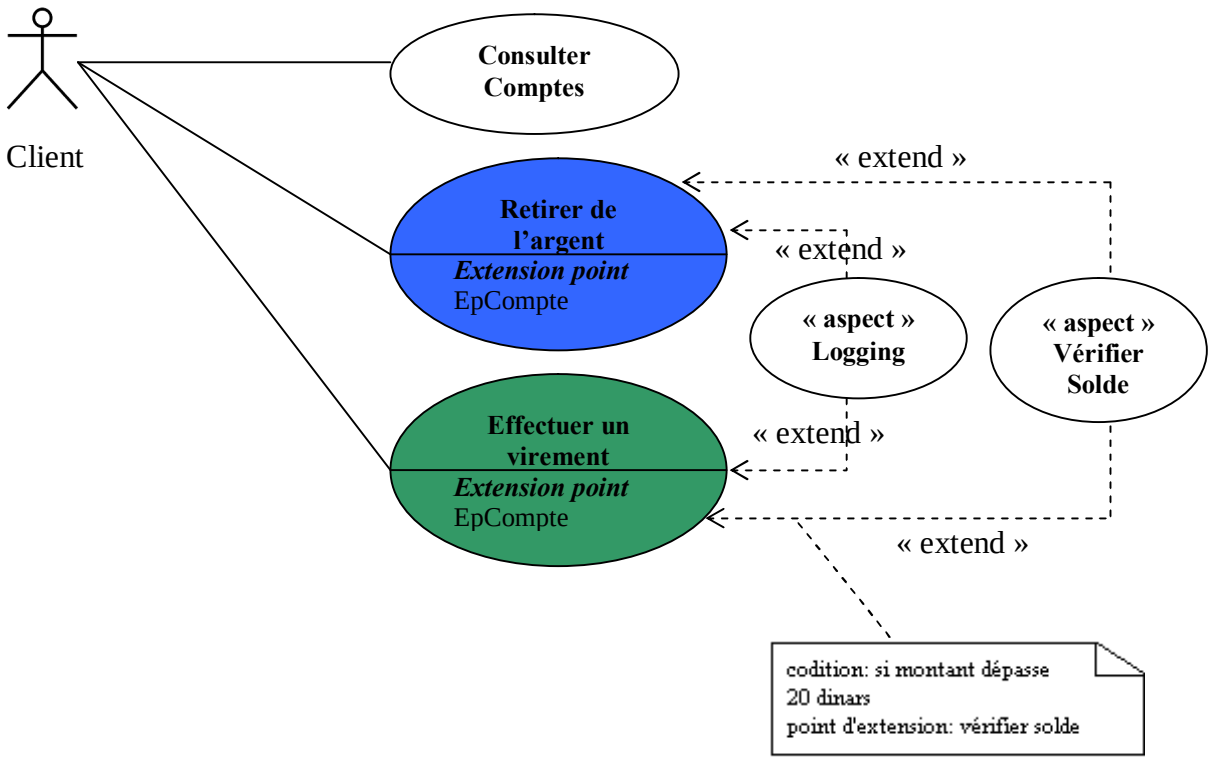


FIG.4.8 Diagramme de cas d'utilisation après l'ajout des aspects

Lôintégration à lôapplication « système bancaire » des deux fonctionnalités auxiliaires *Logging et vérifier solde*, se fait grâce à la notion dôaspects. On introduit deux aspects *Logging et vérifier solde*.

Par conséquent, ces deux aspects entrecoupent les fonctionnalités de base *faire un virement* et *retirer de l'argent*. Dans la vue de cas dôutilisation proposée à la figure, les deux aspects *Logging et vérifier solde*, sont considérés comme des cas dôutilisation dôextension. Pour modéliser le comportement transverse de ces deux aspects et leur insertion dans les cas dôutilisation de base *effectuer un virement* et *retirer de l'argent*, la relation « extend » de UML est utilisée. Les cas dôutilisation de base seront donc modifiés implicitement au point dôextension indiquée « vérifier_solde » nommé *EpCompte*

La figure 4.8 montre le nouveau diagramme après lôajout des fonctionnalités transversales.

classes participantes

Afin dûidentifier les concepts du domaine, nous allons prendre les cas dûutilisation un par un et poser pour chacun la question suivante : quels sont les **concepts qui participent** à ce cas dûutilisation ?

Par exemple, pour les cas qui nous intéresse («Retirer de lûargent») et («Faire un virement»), nous identifions facilement les concepts « Compte» et «Transaction ». Nous allons modéliser ces concepts sous forme de diagrammes de classes contenant uniquement des attributs et des associations.

Le concept de Compte est un concept du domaine, car dans les banques réelles le client doit également avoir un compte avant quûil puisse réaliser ses opérations.

Le **diagramme de classes du domaine** résultant est donné ci-après (figure 4.9).

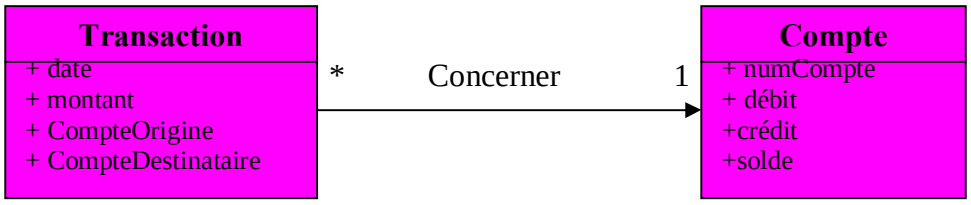


FIG.4.9 Diagramme de classes du domaine

3. 2 Phase de conception :

Cette phase comporte trois étapes :

1. Allouer les comportements aux classes

Dans cette étape, nous allons décrire les différents scénarios des deux cas « retirer de lûargent » et « faire un virement » et attribuer des **responsabilités précises de comportement** aux classes dûanalyse identifiées précédemment. Nous représenterons le résultat de cette étude dans des **diagrammes d'interactions (diagramme de séquence).** Chaque diagramme va ainsi représenter un ensemble dûobjets de classes différentes collaborant dans le cadre dûun scénario dûexécution du système. Tout cela est représenté sur les figures suivantes (diagramme de séquence).

2. Identifier les points de coupure

Comme nous avons vu auparavant ; les cas dûutilisation fonctionnels *retirer de l'argent* et *faire un virement* sont étendus par les aspects *logging* et *vérifier solde* au point dûextension « vérifier_solde » nommé *EpCompte*. Ce dernier sera transformé en un point de coupure

portant le même nom et modélisé par une interface stéréotypée *« PointCut »* au niveau de diagramme de classes.

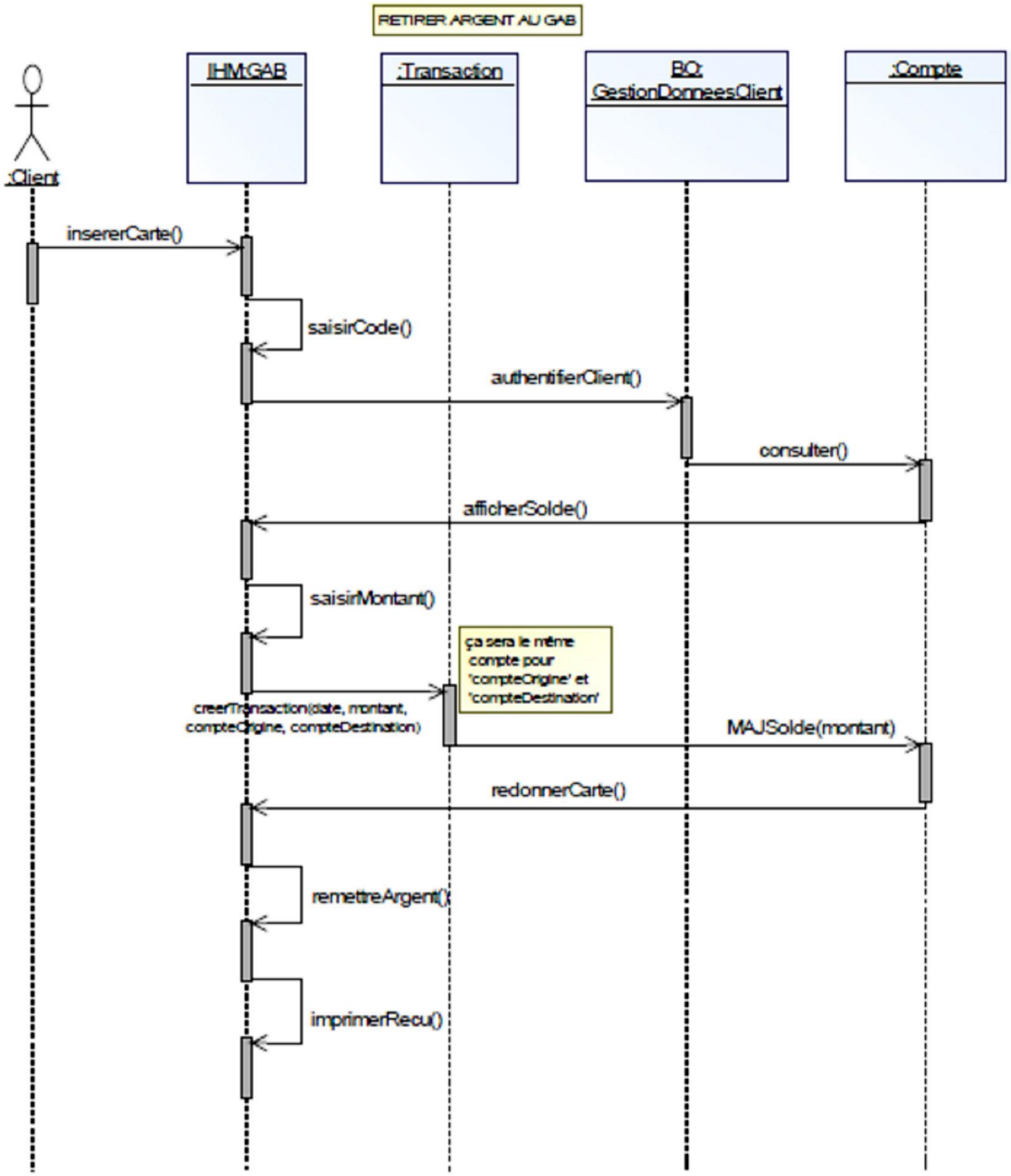


***FIG.4.10 Diagramme de séquence correspondant au CU* « Retirer Argent au GAB »**

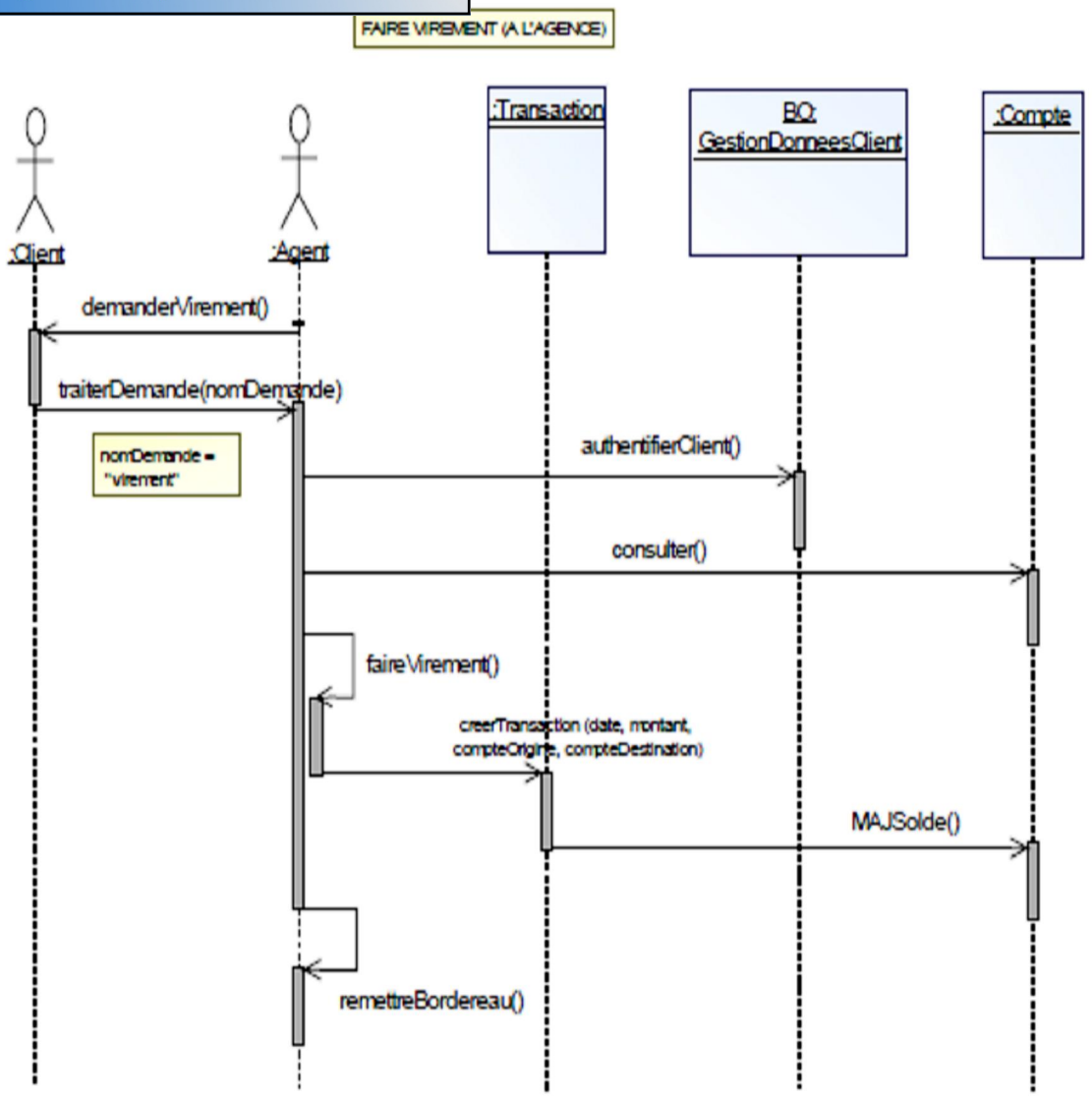


FIG.4.11 Diagramme de séquence correspondant au CU « **Faire Virement (à l'agence)** »

1. Etablir le diagramme de classes

En partant du modèle du domaine et des classes participantes, nous allons affiner et compléter le diagramme de classes. Pour cela nous utiliserons les diagrammes dôinteractions que nous venons de réaliser pour :

- ✓ Ajouter ou préciser les **opérations** dans les classes (un message ne peut être reçu par un objet que si sa classe a déclaré lôopération publique correspondante).
- ✓ Ajouter des **types** aux attributs et aux paramètres et retours des opérations.
- ✓ Affiner les **relations** entre classes : association, généralisations ou **dépendances**.

eption résultant est fourni par la figure 4.12.

Considérons les cas dôutilisation *Logging et vérifier solde* de notre exemple. La figure 4.13 montre comment les exigences de ces fonctionnalités sont intégrées dans le diagramme de classes UML. Les comportements des cas dôutilisation *Logging et vérifier solde* sont décrits par des classes nommées respectivement *Logging et vérifier solde* portant le stéréotype « aspect ». Quant au point dôextension *EpCompte* , il correspond à un point de coupure et est donc modélisé par une interface portant le stéréotype « Pointcut ». Un point de coupure est lié à un ensemble de points de jointure. Une relation de dépendance « crosscut » relie donc lôinterface « Pointcut » à ses points de jointure. Lôinterface *EpCompte* a une relation « crosscut » pointant respectivement vers les points de jointure correspondant à la méthode *créditer ()* et à la méthode *débiter ()* de la classe *Compte.*

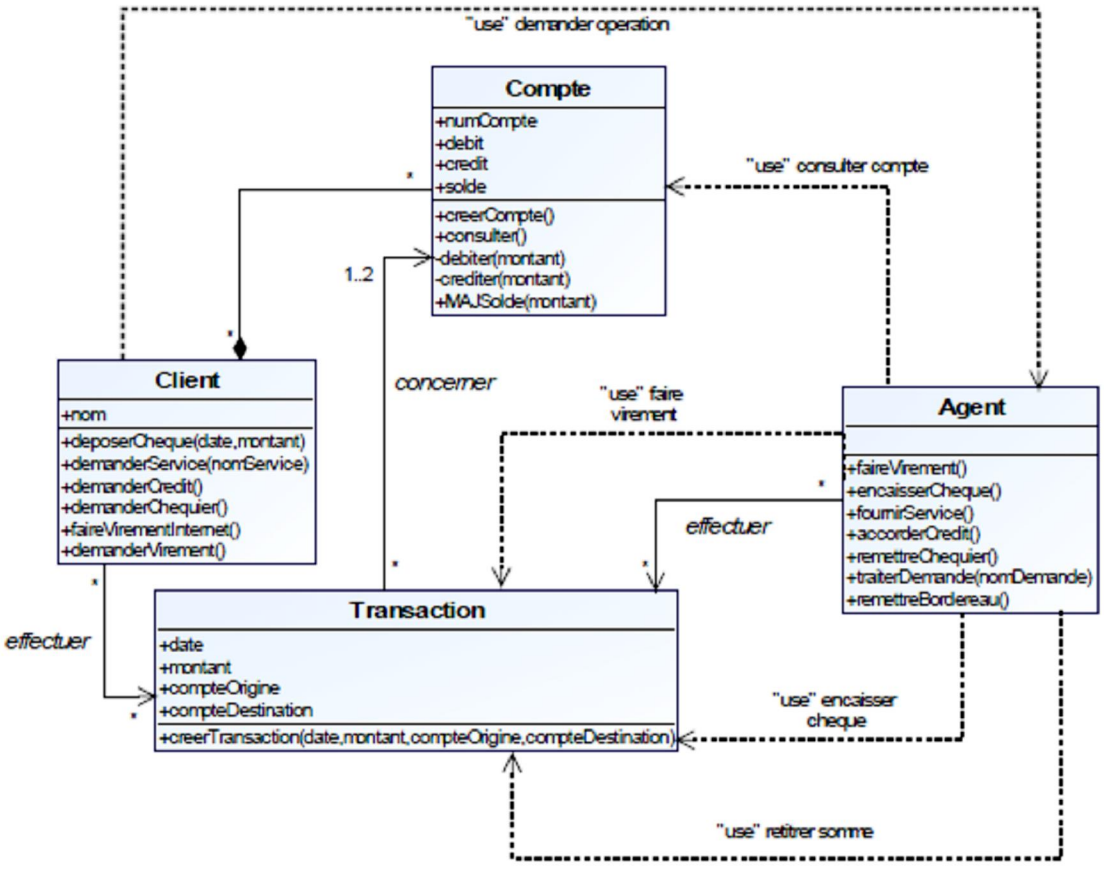


FiG.4.12 Diagramme de classes des CU Gestion des comptes

Interface EpCompte contient une opération abstraite nommée respectivement *EpCompte* qui doit être exécutée quand un de ses points de jointure est atteint. Lôaspect *Logging* implémente lôinterface *EpCompte* et propose donc une méthode, appelée advice, qui sera exécutée aux points de jointure spécifiés. Une relation de réalisation lie la classe « aspect » à lôinterface « pointcut ». À remarquer quôun advice est annoté avec lôun des stéréotypes « before », « after » ou « around » selon quôil est exécuté respectivement avant, après ou autour des points de jointure référencés par le point de coupure. La figure montre lôintégration des deux aspects au niveau de diagramme de classes.

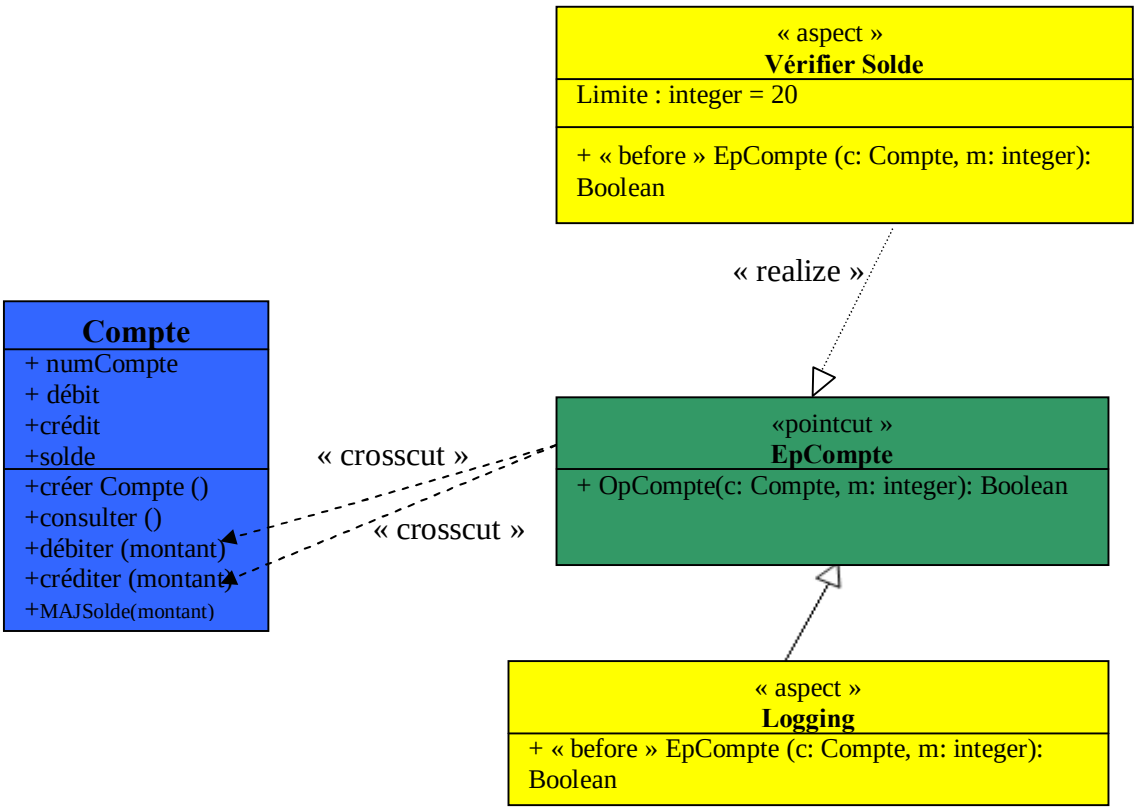


FIG.4.13 **Diagramme de classes après l'ajout des aspects**

3.3 Implémentation de quelques aspects et exemples

Une fois les choix techniques définis nous pouvons donner un aperçu de la cartographie technique cible :

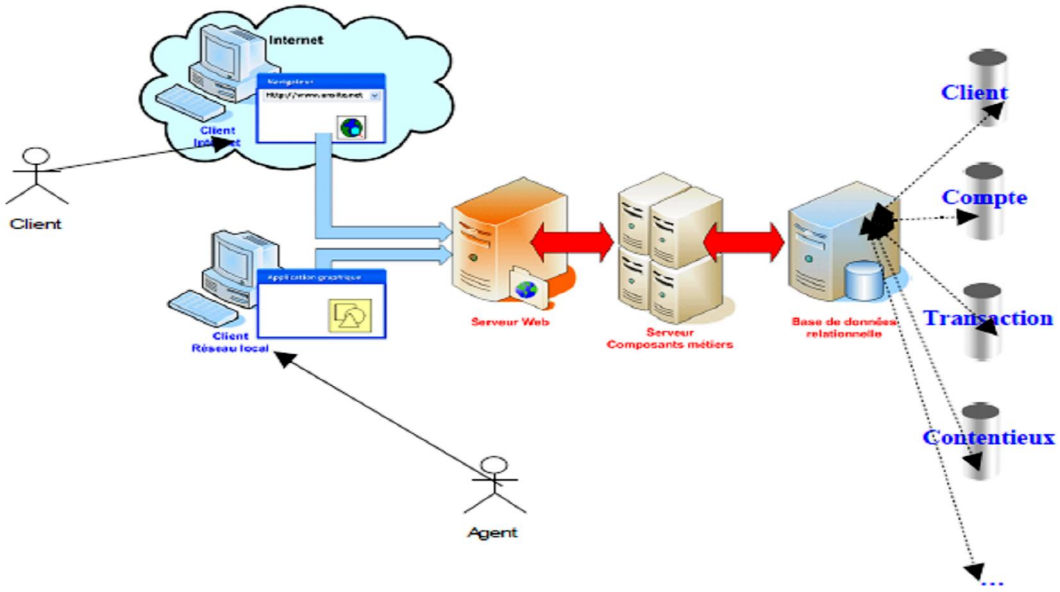


FIG.4.14 Architecture globale du système bancaire

1. Pré requis : afin dôimplémenter notre application nous avons besoin de ce qui suit :

- **installation du JDK :** Le JDK (**Java Development Kit** en anglais, Kit de Développement Java en Français) représente lôoutillage indispensable au développeur Java. Ce kit contient les outils nécessaires **programmer** en java, **exécuter** ses programmes java, **tester** ses programmes java et **livrer** ses programmes java à ses clients.
- **Installation kit de développement Aspectj (AJDK) : outil** indispensable pour programmer avec des aspects AspectJ (voir Annexe A).

2. Objectifs :

- Créer et mettre en oeuvre un aspect de LOG
- Déclarer un aspect, avec AspectJ
- Déclarer une coupe, un point de jonction avec AspectJ
- Comprendre la valeur ajoutée de la programmation Aspects

3. Programme :

Le programme est divisé en deux parties :

- Partie 1 : implémenter les retraits dôargent sans aspectj.
- Partie2 : implémenter les retraits dôargent avec un aspect AspectJ de log : **LogAspect.aj**

ectj

Fichiers à créer :

- Créer un fichier **CompteBancaire.java** qui sera la classe représentant un compte bancaire. Ajouter une propriété solde ainsi que getter/setter puis constructeur. Ajouter enfin les méthodes de retrait et dépôt.

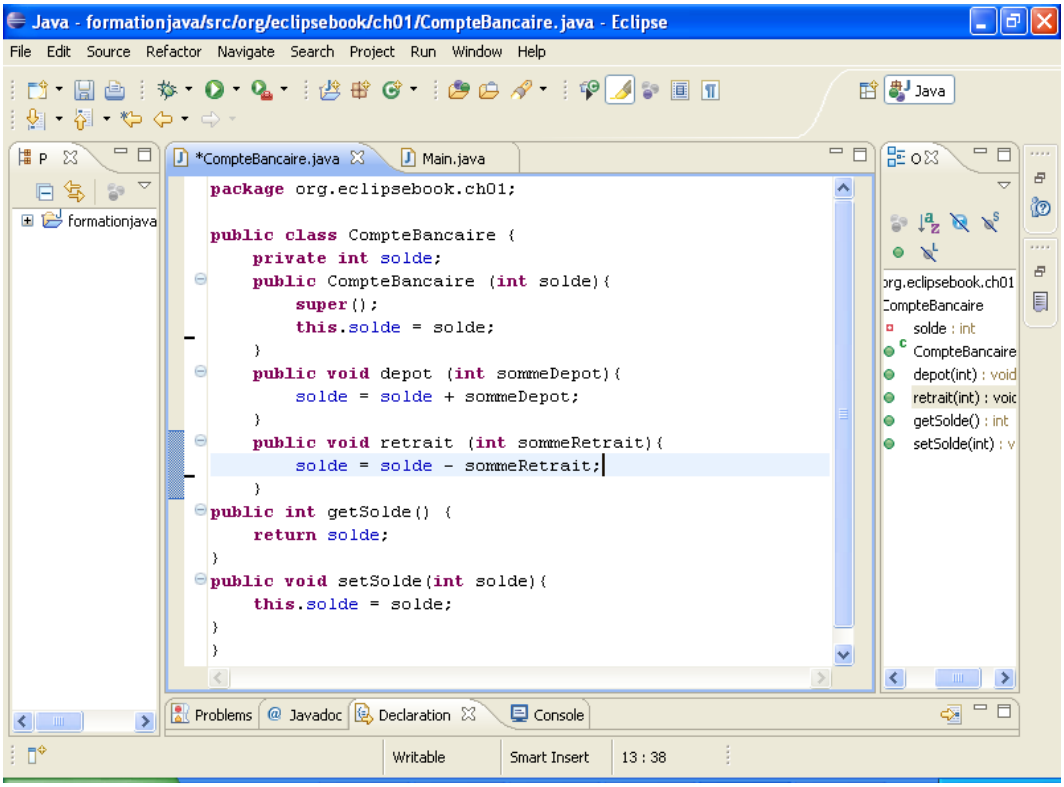


FIG.4.15 Création de la classe CompteBancaire.Java

- Créer un fichier **Main.java** qui sera la classe principale de lôapplication (il contiendra la méthode main()). Elle va instancier un compte (Figure 4.16)

En ce qui concerne les 4 étapes ci-dessus, nous remarquons que :
ð 2 étapes ne concernent pas directement le ômétierô. En effet les étapes 2 et 4 sont liées à une préoccupation ôtechniqueô: celle de tracer un évènement (ici le retrait).
ð 2 étapes concernent le métier. En effet les étapes 1 et 3 sont liées à des préoccupations directement liées au métier bancaire :

Le ôtisseur dôaspectsô AspectJ va nous permettre par la suite dôisoler cette préoccupation technique dans un fichier distinct : le fichier LogAspectj.aj qui représente lôaspect LOG.

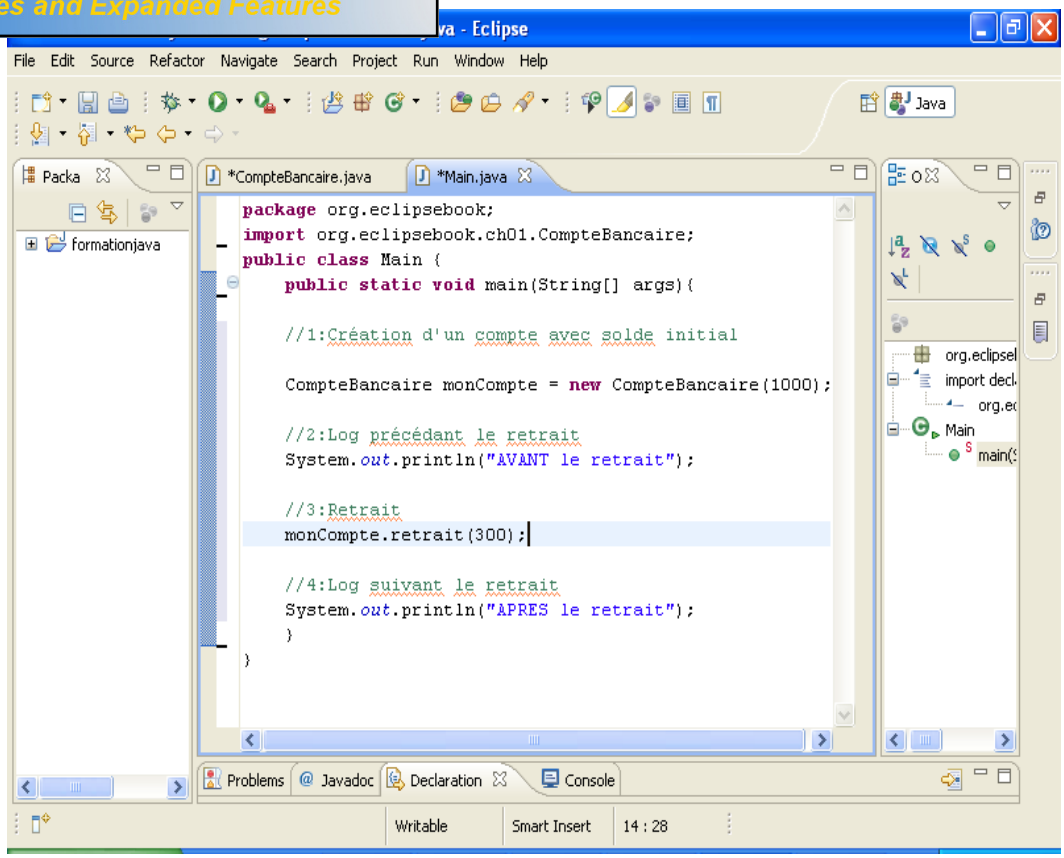


FIG.4.16 Création de la classe principale Main.Java

- Lancer lêxécution de la classe principale : Main.Java

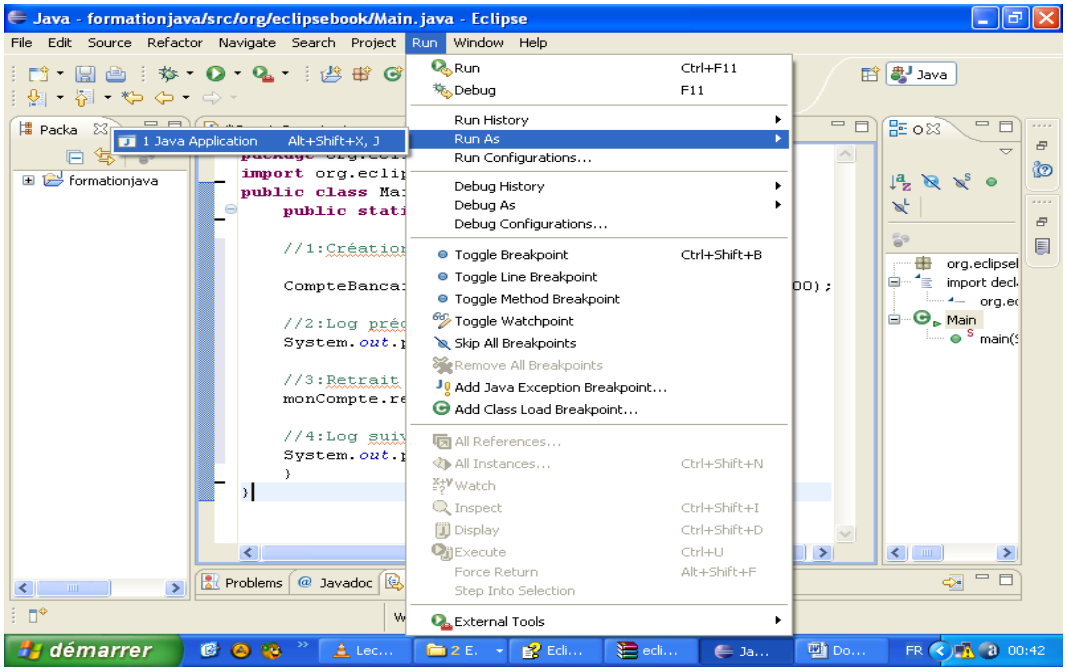


FIG.4.17 Exécution de la classe principale Main.Java

dent et suivent lôappel à la méthode retrait().

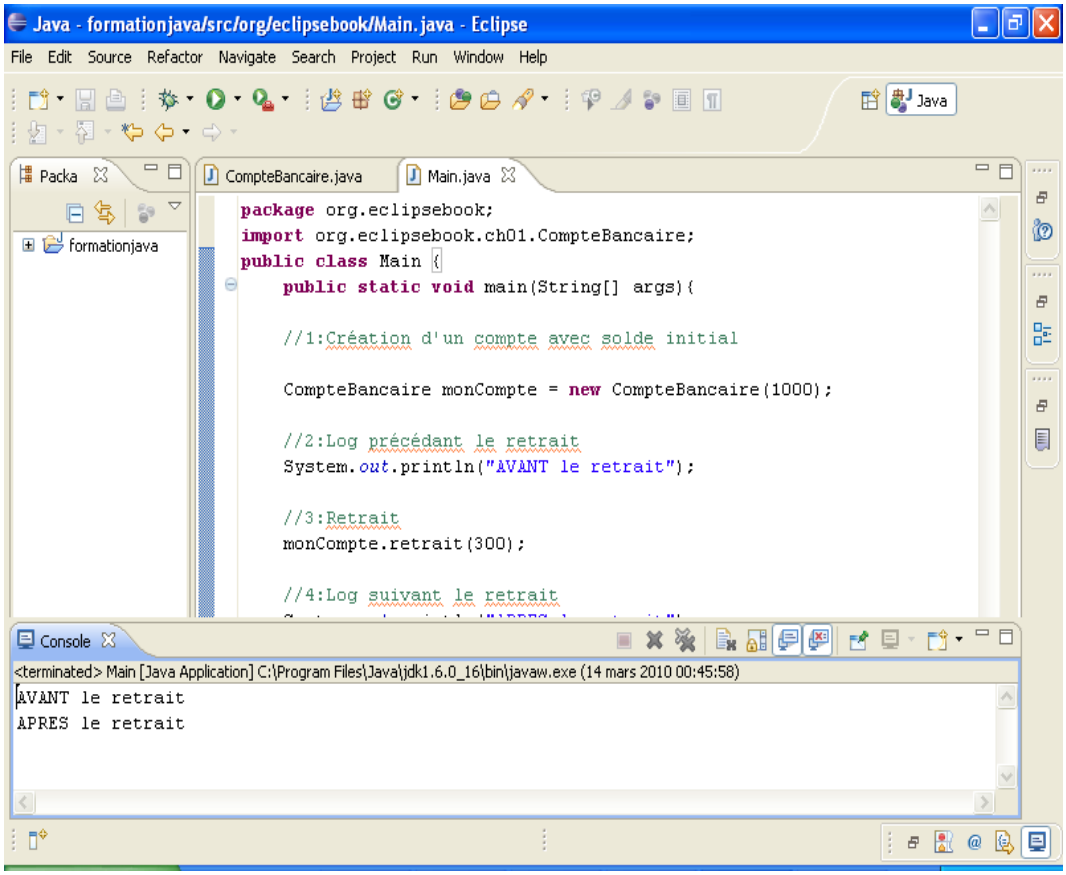


FIG.4.18 Résultat de l'exécution de la classe principale Main.Java

Pour cela nous avons du écrire dans la classe cliente les lignes suivantes :

- System.out.println("AVANT le retrait") ;
- System.out.println("APRES le retrait") ;

Nous aurions également pu écrire ces ligne dans la méthode retrait() de la classe appelée (CompteBancaire).

Cêst là quêintervient un tisseur dôaspect comme AspectJ

3.2 Partie 2 : implémentation avec aspectJ

Dans cette partie nous allons mettre le tisseur dôaspect AspectJ en action. Nous allons supprimer tout code de Log dans nos classes et centraliser la gestion des logs dans un aspect aspectJ : **LogAspect.aj**

Nous allons utiliser le compilateur ajc (surcouche du compilateur javac) pour compiler aussi bien lôaspect **LogAspect.aj** que les classes **Main.java** et **CompteBancaire.java**.

n de la façon suivante :

Main.java

```
package com.objis.demoaspectj;

import com.objis.demoaspectj.banque.CompteBancaire;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {

        // 1 : Création d'un compte avec solde initial
        CompteBancaire monCompte = new CompteBancaire(1000);

        // 2 : Retrait
        monCompte.retrait(300);

    }

}
```

FIG.4.19 Modification de la classe principale Main.java

Comme nous le constatons, il nôy a aucune ligne associée au Log . Côest un aspect LogAspect.aj que nous allons créer qui va õintercepterô toute demande de retrait.

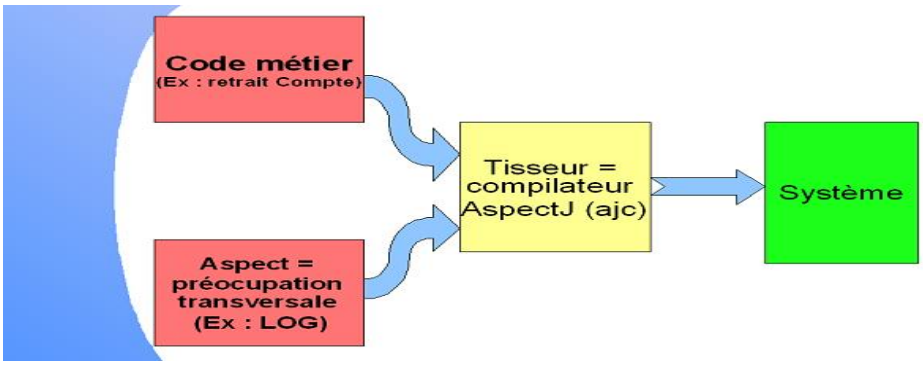


FIG.4. Tissage d'aspect avec Aspectj

Créer un fichier **LogAspect.aj** et ajouter le contenu suivant :

❖ **Exemple d'aspect Aspectj : LogAspect.aj**

```
package com.objis.demoaspectj.aspects;

public aspect LogAspect {                                    (1)

(2) pointcut logRetrait()                                    (3)

: execution(* com.objis.demoaspectj.banque.CompteBancaire.retrait(..));
```

```
4   System.out.println("AVANT le retrait");
    }

    after() : logRetrait() {
    System.out.println("APRES le retrait");   5
    }

}
```

❖ **Explication :** les différentes parties du code présenté ci-dessus sont expliquées comme suit :

ð 1 : nous déclarons un **aspect** à travers le mot clé ôaspectô. Ici lôaspect LogAspect.

ð 2 : nous déclarons une **coupe** nommée logRetrait() à travers le mot clé ôpointcutô. Une coupe est un ensemble de **point de jonction** (Moments dôexécution où il se passe quelque chose qui nous intéresse. Côest lôéquivalent de point dôarrêt lors dôun débogage)

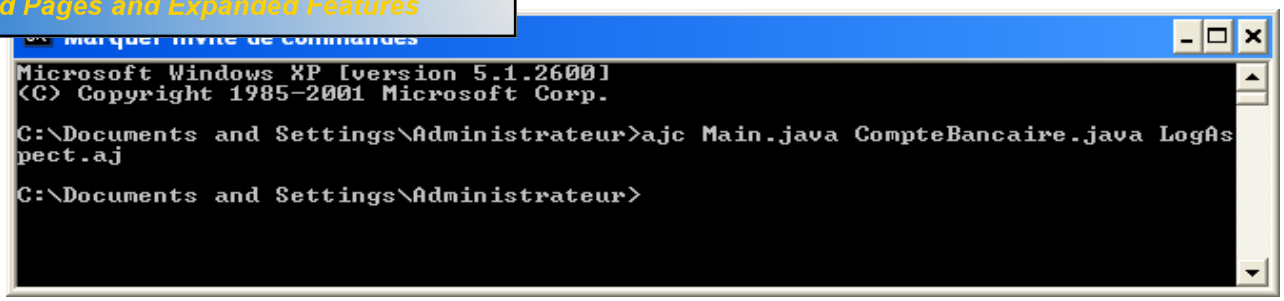
ð 3 : nous déclarons lôensemble des points de jonction. Ici toute méthode retrait() de la classe com.objis.demoaspectj.banque.CompteBancaire, quelque soit le nombre de paramètre de la méthode retrait (..) et quelque soit le type de retour (*) de la méthode retrait().

ð 4 : nous déclarons un **greffon** type before() : le code System.out.println("AVANT le retrait") ; sera lancé juste avant tout point de jonction (c à d ici toute exécution de la méthode retrait())

ð 5 : nous déclarons un greffon type after() : le code System.out.println("APRES le retrait") ; sera lancé juste après tout point de jonction (c-à-d ici toute exécution de la méthode retrait())

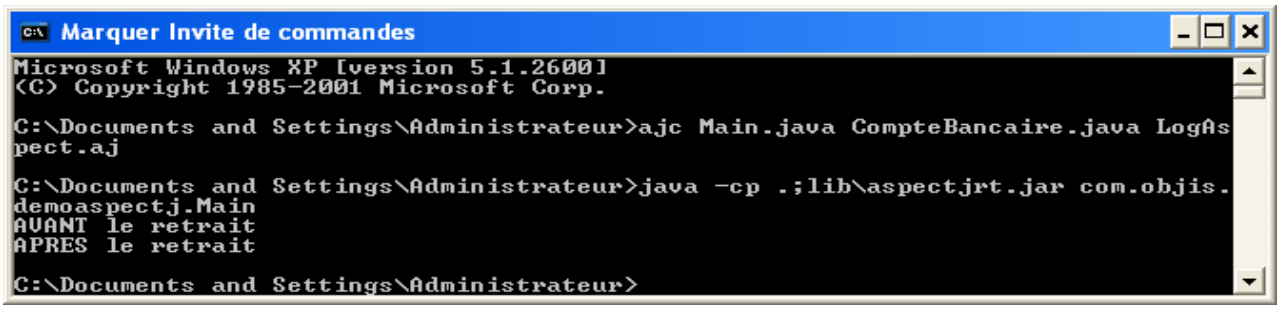
Nous avons codé notre premier aspect 100% aspectJ. Il reste à le compiler.

▸ Compilons lôensemble des classes Main.java, CompteBancaire.java et LogAspect.aj en utilisant le compilateur ajc (aspectj compiler) installé (voir Annexe A).



Nous obtenons ceci :

- Lancer lôexécution de la classe Main :



Le résultat est le même que dans la partie 1. Mais le code de notre classe principale (Main.java) est plus léger. Nous nous sommes concentrés sur le métier et non sur une préoccupation de log.

4. Évaluation du processus

Nous évaluons le processus affiné avec la méthode FOCSAAM vue dans le chapitre 1, section 6.2 en répondant aux questions formulées dans le tableau 4.1.

Nous avons proposé une approche pour construire une architecture basée sur la notion Aspect et orientée utilisateur. En utilisant les caractéristiques du domaine et avec identification hâtive des besoins fonctionnels et non fonctionnels du système, nous pouvons proposer dès le début du développement le style du système.

mière configuration proposée en utilisant l'étude de domaine tout en considérant les propriétés non fonctionnelles exigées. Cette première configuration permet d'introduire les différents systèmes intervenants dès le début.

L'architecte doit trouver un bon point de départ, l'architecture de base ou une première configuration documentée avec les propriétés non fonctionnelles, pour justifier ses choix et pour rendre son expérience disponible aux autres.

Le tableau 4.1 montre le résumé de l'évaluation de la méthode.

Eléments de FOC-SAAM	**Méthode de Développement Basée sur la notion d'Aspect et Orientée Utilisateur**
Définition de l'architecture	Définition standard
Objectif de méthode	Architecture guidée par les aspects et orientée utilisateur
Attributs de qualité	Pas spécifiés
Etapes applicables	Analyse, conception
Entrée/Sortie	Entrée : le cahier des charges (les besoins fonctionnels et non fonctionnels du système et les contraintes de développement). Sortie : la conception de l'architecture du Système.
Domaine d'application	Pas de contraintes.
Bénéfices	La prise en charge des propriétés non fonctionnelles est garantie.
Intervenants	analystes, architectes, client
Support de processus	Suffisamment détaillé
Aspects sociaux	Pas de directives
Ressources exigées	Pas spécifié
Activités de méthode	Deux phases : 1ère avec 3 activités (pas d'ordre spécifique) et 2ième avec 2 étapes séquentielles.
Description architecturale	Composant/connecteurs
Approches d'évaluation	Scénarios de qualité
Outil support	Java, aspectj
Niveau de maturité	Recherche
Validation de méthode	Utilisée dans l'étude de cas : Gestion d'un compte bancaire

TAB. 4.1 Évaluation de la méthode

5. Conclusion

Dans ce chapitre nous avons appliqué notre méthode sur une étude de cas : un système de gestion dôun compte bancaire. Cette étude de cas nous a permis de valider notre méthode. La réalisation de tels systèmes nécessite une conception orientée par les qualités ou propriétés non fonctionnelles et les cas dôutilisation (centrée utilisateur).

Une étude des systèmes existants montre qu'ils sont pour la plupart basés sur des architectures hybrides : orientées utilisateur et orientées aspects. Nous avons utilisé notre méthode de développement permettant de prendre en compte une telle architecture hybride.

Nous avons défini le cahier de charge d'un système de gestion dôun compte bancaire et fait l'analyse et la conception de ce dernier, ensuite nous avons créé et mis en î uvre lôaspect LOG pour montrer la valeur importante ajoutée par la programmation orientée aspect.

Nous avons terminé ce chapitre par lôévaluation de notre méthode avec la méthode **FOC-SAAM**.

Conclusion Générale et Perspectives

Dans cette thèse nous avons étudié les problèmes liés aux méthodes de développement des logiciels en particulier les problèmes liés aux exigences non fonctionnelles. Nos contributions principales se résument dans les points suivants :

- Nous avons proposé une méthode de développement de logiciels:
  - cette méthode est guidée par les propriétés non fonctionnelles (basée sur la notion dôaspect). Dès la phase d'analyse, la méthode relie les propriétés non fonctionnelles et les besoins fonctionnels du système. Les besoins non fonctionnels sont considérés comme des aspects.

    L'étude des propriétés non fonctionnelles du système guide les concepteurs pendant le développement. Les décisions architecturales qui sont prises pour répondre aux exigences fonctionnelles et/ou non fonctionnelles du système, doivent respecter la notion de qualité. Ainsi le système en cours de développement pourra répondre à certains critères de qualité dôun bon logiciel.
  - cette méthode et orientée utilisateur : La principale qualité dôun logiciel étant son utilité, cêst-à-dire son adéquation avec les besoins des utilisateurs. Toutes les étapes, de la spécification des besoins, doivent être guidées par les cas dôutilisation qui modélisent justement les besoins des utilisateurs. Cette méthode montre bien lôimportance de la technique des cas dôutilisation dans lôingénierie des systèmes.
  - cette méthode trace bien les propriétés fonctionnelles et non fonctionnelles :

    Toutes les décisions architecturales (les raffinements) sont enregistrées sous forme de couples problème-solution. Ainsi toutes les propriétés non fonctionnelles du système et les solutions introduites pour les satisfaire seront documentées et bien tracées.
  - cette méthode intègre une méthode d'évaluation : elle permet de comparer à chaque étape de raffinement, la relation entre les qualités du système en cours de développement et celui attendu.
- Nous avons proposé un profil UML permettant de décrire les principaux concepts du paradigme aspects. En intégrant la notion dôaspect non seulement dans les diagrammes de classes mais également dans les diagrammes de cas dôutilisation. Notre approche

les fonctionnalités de nature transversale dès les ion des besoins.

Ce travail nous a permis dôavoir de nouvelles perspectives à savoir :

- Développer une solution concrète pour séparer les diverses préoccupations montrées dans le papier selon le paradigme « services » et « aspects » pour montrer la pertinence de notre approche.
- Proposer une méthode de développement basée sur les patterns : la méthode aide et simplifie l'utilisation des patterns définis comme des couples <problème-solution> [35], décrits en UML 2.0 et les décisions architecturales sont guidées par les propriétés non fonctionnelles.
- Inclure le paradigme aspect dans dôautres diagrammes tel que le diagramme dôétats et définir une méthodologie de vérification des modèles par aspects.

Introduction générale

ntielle et elle envahit tous les domaines de la vie quotidienne. La taille et la complexité des logiciels augmentent et il y a de plus en plus besoin de systèmes décentralisés et coopératifs. Le développement des logiciels de grande taille fait intervenir différents spécialistes dans le cycle de vie du logiciel : des concepteurs, des architectes, des développeurs, des testeurs et des équipes de maintenance qui doivent communiquer entre eux et manipuler un même artefact. De plus les logiciels évoluent dans le temps et leur développement est devenu coûteux et complexe.

La diminution du coût et la complexité de développement et de maintenance des logiciels nécessite d'avoir des logiciels à couplage faible, maintenables et faciles à réutiliser. La réutilisation réduit les étapes de conception et de test des composants réutilisés. Elle permet la maintenance et l'évaluation des applications [27]. Ainsi le logiciel doit satisfaire des exigences non fonctionnelles et s'adapter aux besoins des clients.

Le travail de développement de l'architecture devient crucial et les chercheurs essaient de la rendre explicite, bien formée et maintenable [28].

La séparation des préoccupations (SoC, Separation of Concerns) est un concept présent depuis de nombreuses années dans l'ingénierie des logiciels [36,37]. Les différentes préoccupations des concepteurs apparaissent comme les premières motivations pour organiser et décomposer une application en un ensemble d'éléments compréhensibles et facilement manipulables. La séparation en préoccupations apparaît dans les différentes étapes du cycle de vie du logiciel et sont donc de différents ordres. Il peut s'agir de préoccupations d'ordre fonctionnel (séparation des fonctions de l'application), technique (séparation des propriétés du logiciel système).

Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais par parties. Cette approche réduit la complexité de conception, de réalisation, mais aussi de maintenance d'un logiciel et en améliore la compréhension, la réutilisation et l'évolution. Cette séparation est la motivation d'approches telles que la programmation par aspects [38], les filtres de composition [40], la programmation adaptative [39, 41, 42, 43] ou encore la programmation par sujet [44, 45, 46,47].

L'une des techniques les plus étudiées à l'heure actuelle correspond à la programmation orientée aspect (POA), (AOP, Aspect-Oriented Programming). Celle-ci a été

ROX PARC[1] en 1997 [38]. Il s'agit d'une technique
ons complexes, telles que les applications distribuées.

Elle se fonde sur une séparation claire entre les préoccupations « métiers » (ou « fonctionnelles ») et « non fonctionnelles » présentes dans les applications.

Néanmoins, la POA ne s'arrête pas à ce premier niveau de découpage, et vise à appliquer la séparation à toutes les préoccupations, qu'elles soient fonctionnelles ou non. Chaque aspect est destiné à être développé de façon indépendante puis intégré à une application par un processus dit de tissage d'aspects (aspect weaving). L'une des expérimentations les plus abouties du langage orienté aspect est AspectJ [48] développé par l'équipe a l'origine de la POA.

De ce fait, pour les auteurs de la méthodologie POA, une bonne partie de la complexité et du manque de robustesse des systèmes existants provient du fait que la mise en oeuvre des fonctionnalités orthogonales, ou transversales, au langage s'entrelace avec le reste du code de l'application.

Par ailleurs, les cas d'utilisation d'UML [52] (Unified Modeling Language) - que l'on peut traduire par "langage de modélisation unifié"- représentent un élément essentiel de la modélisation orientée objet. Ils interviennent très tôt dans la conception, et doivent en principe permettre de concevoir, et de construire un système adapté aux besoins de l'utilisateur (construire le bon système). Ils doivent également servir de fil rouge tout au long du développement, lors de l'analyse, de conception, d'implémentation et de tests. Ils servent donc aussi bien à définir le produit à développer, à modéliser le produit, qu'à tester le produit réalisé.

L'architecture logicielle a des étapes et des aspects différents. Une des phases les plus importantes de l'architecture est la phase de conception. Les décisions prises dans cette phase sont très décisives et coûteuses en cas de changement. Nous nous intéressons aux processus de conception et de développement de logiciel. Une méthode de conception architecturale doit avoir plusieurs propriétés parmi lesquelles, on peut trouver :

- **Analyse dirigée par les cas d'utilisation :** La principale qualité d'un logiciel étant son utilité, c'est-à-dire son adéquation avec les besoins des utilisateurs, toutes les étapes, de la spécification des besoins à la maintenance, doivent être guidées par les cas d'utilisation qui modélisent justement les besoins des utilisateurs.

[1] Le PARC (Palo Alto Research Center) a entre autre gratifié avant la programmation orientée aspects de Smalltalk, du réseau d'entreprise, de l'interface graphique et de la souris (mise en oeuvre des idées novatrices de Engelbart), de l'imprimante Postscript laser et du premier ordinateur personnel, et ce dès la fin des années soixante.

és non fonctionnelles : certaines des propriétés non sidérées dès la phase dôacquisition des besoins. Elles imposent des contraintes architecturales au système et leur prise en charge doit être incorporée dans l'architecture. Une fois la conception terminée, si ces propriétés ne sont pas acquises, le coût de changement de l'architecture peut être considérable.

- **Réutilisation d'éléments de conception existants :** il existe différentes manières de réutiliser les éléments architecturaux comme l'ingénierie de domaine, les bibliothèques de codes, les patterns, etc. Les patterns sont un moyen de réutilisation, applicable dès la phase de conception, pourtant leur utilisation est limitée par des problèmes de sélection.
- **Facilité d'application :** la méthode doit laisser l'analyste suivre aussi son intuition tout en le guidant pour qu'il justifie ses choix. La méthode doit surtout aider l'analyste quand il ne sait pas faire, mais ne pas le contraindre.
- **Traçabilité des décisions architecturales, leurs raisons d'être et leurs impacts :** différentes personnes interviennent pendant le cycle de vie d'un logiciel, comme par exemple, lôutilisateur ou client, l'équipe d'analystes, l'équipe de développement, l'équipe de maintenance, etc. Pour que les décisions soient compréhensibles pour tout le monde, et afin de rester objectifs, ces décisions doivent être documentées avec les raisons pour lesquelles elles sont prises, ainsi que leurs impacts sur l'architecture.
- **Diminution du gap entre l'analyse de besoins et le développement :** il y a souvent un vide entre la phase d'analyse des besoins effectuée en général en langage naturel et la première configuration du système en général décrit de manière semi formelle (UML). La méthode de conception doit remontrer le résultat de l'analyse pendant conception.

À notre connaissance les méthodes de conception sont souvent centrées sur quelques unes des propriétés présentées ici et elles ne proposent pas de directives pour les autres.
Parmi ces propriétés la prise en compte des propriétés non fonctionnelles et lôadéquation avec les besoins des utilisateurs sont les plus importantes. Ces dernières années, plusieurs travaux proposent des processus et des méthodes qui sont guidés par une ou plusieurs propriétés non fonctionnelles. Ces méthodes n'ont pas encore atteint leur maturité.

L'architecture logicielle d'un système permet de prendre en compte en outre les besoins fonctionnels, les objectifs de qualité ou besoins non fonctionnels du système à développer. La

ycle de vie d'un système, de ces aspects est essentielle qualité. La décomposition fonctionnelle d'un système en général est réalisée en utilisant des méthodes classiques de développement de logiciel. Cependant il n'y a aucune méthode fiable considérant la décomposition non fonctionnelle.

Dans ce travail, notre objectif est de proposer une méthode de développement qui garantit la prise en compte de toutes les propriétés notamment les propriétés non fonctionnelles. L'architecture se définit en considérant d'une part une spécification des besoins non fonctionnels du domaine qui seront considérés comme des aspects et d'autre part les besoins fonctionnels du système définis par des cas d'utilisation (vision orientée utilisateur). La configuration initiale est raffinée en introduisant des nouveaux éléments, répondant aux propriétés non fonctionnelles spécifiques. Notre contribution réside dans le fait de rendre explicites toutes les décisions architecturales, y compris celles concernant la prise en compte des besoins non fonctionnels.

La première étape de notre travail consiste à proposer une approche permettant de décrire les principaux concepts du paradigme aspect. Cette dernière est un profil UML. En intégrant la notion d'aspect non seulement dans les diagrammes de classes mais également dans les diagrammes de cas d'utilisation, notre approche permet de prendre en compte les fonctionnalités de nature transversale dès les premières étapes de la spécification des besoins. Notre approche est de nature itérative et procède donc par raffinements successifs des modèles.

Dans une deuxième étape, nous proposons une méthode de développement de logiciels basée sur la notion d'aspect qui représente les préoccupations de nature transversale et d'une autre part dirigée par les cas d'utilisation qui explosent bien les besoins des utilisateurs. Cette méthode se base sur l'approche proposée dans la première étape.

Ce rapport de thèse est organisé comme suit :

- ✓ Le chapitre 1 présente les concepts de base de la programmation orientée aspect, à commencer par les limites de la programmation orientée objet, nous allons définir ensuite le concept aspect. Sont également abordées les notions connexes de coupe, de code advice, de point de jonction et d'introduction. Ces concepts sont présentés indépendamment de tout langage et de outil. Nous allons par la suite nous intéresser à réalisation de ce concept en pratique avec le langage AspectJ. Nous terminerons ce premier chapitre par l'apport du paradigme aspect au niveau de la phase de conception.

les différentes préoccupations des utilisateurs et leurs
lôUML. Nous commencerons par la définition des cas
dôutilisation, les différentes relations qui les relient et comment les utiliser pour représenter les besoins et les exigences des utilisateurs. Nous terminerons par montrer la place des cas dôutilisation dans le développement.

- ✓ Le chapitre 3 propose une méthode de développement de logiciels. Nous définissons un profil UML qui permet de décrire les principaux concepts du paradigme aspect. Ce profil permet de prendre en compte les fonctionnalités de nature transversale dès les premières étapes de la spécification des besoins. Ensuite nous terminerons par la présentation du processus de notre méthode de développement. Ce dernier est basé sur les concepts du paradigme aspect et dirigé par les cas dôutilisation dôUML (orienté utilisateur).
- ✓ Le chapitre 4 présente lôapplication de ce processus à un cas particulier relatif au développement dôun « Système de Gestion dôun Compte Bancaire » afin de valider notre solution conceptuelle.

Nous proposons ensuite la partie consacrée à la conclusion générale dans laquelle quelques perspectives seront fournies.

A la fin de ce mémoire nous trouverons des annexes présentant les différents sujets non étoffés dans les chapitres principaux de cette thèse.

...es Tableaux

ystème permet de prendre en compte en outre les besoins fonctionnels, les objectifs de qualité ou besoins non fonctionnels du système à développer. La prise en compte dès le début, dans le cycle de vie d'un système, de ces aspects est essentielle pour le développement des systèmes de qualité. La décomposition fonctionnelle d'un système en général est réalisée en utilisant des méthodes classiques de développement de logiciel. Cependant il n'y a aucune méthode fiable considérant la décomposition non fonctionnelle.

Dans ce travail, notre objectif est de proposer une méthode de développement qui garantit la prise en compte de toutes les propriétés notamment les propriétés non fonctionnelles. L'architecture se définit en considérant d'une part une spécification des besoins non fonctionnels du domaine qui seront considérés comme des aspects et d'autre part les besoins fonctionnels du système définis par des cas dôutilisation (vision orientée utilisateur). La configuration initiale est raffinée en introduisant des nouveaux éléments, répondant aux propriétés non fonctionnelles spécifiques. Notre contribution réside dans le fait de rendre explicites toutes les décisions architecturales, y compris celles concernant la prise en compte des besoins non fonctionnels. Cette méthode est appliquée à l'analyse et la conception dôun système de gestion de compte bancaire.

Abstract

System architecture is constructed considering functional and non-functional requirements, entailing the fulfilment of precise quality goals. The early consideration of these aspects in the life cycle of software is crucial for the development of quality systems. The functional decomposition of a system is in general achieved following classic software development methods. However there are no mature methods considering the non-functional decomposition.

In this work we propose an approach that takes into account all the properties, especially the non-functional requirements. The architecture is derived considering on one hand the domainôs specification of non-functional requirements considered as aspects and on the other hand the system's requirements specified by the Use-Cases (user oriented vision). The initial configuration is completed by introducing new elements, responding to specific non-functional properties. The main contribution of our approach is to explicit all the decisions and their rational on an existing architecture or while constructing a non-functional driven base-line architecture. The approach is applied to analysis and conception of a system bank account management.

الملخص

الهندسة البرمجية لنظام معين،تتطلب الأخذ بعين الاعتبار، علاوة عن الاحتياجات العملية، الأهداف الخاصة بالجودة أو ما يسمى بالاحتياجات غير العملية الخاصة بالنظام المطور. الاهتمام منذ البداية لهذه الجوانب أثناء دورة حياة أي نظام، تعتبر هامة جدا لإنشاء نظام ذو جودة

التحليل العملي لأي نظام في العموم ناتج في الغالب باستعمال طرق كلاسيكية لتطوير البرامج، غير أنه ليس هناك أي طريقة فعالة تأخذ بعين الاعتبار التحليل غير العملي.

أثناء هذا العمل، هدفنا هو عرض طريقة لإنشاء البرامج التي تضمن الأخذ بعين الاعتبار كل الخصائص لا سيما الخصائص غير العملية.

يعرف الهيكل بتحديد من جهة أولى الاحتياجات غير العملية التي تعتبر كجوانب و من جهة أخرى الاحتياجات العملية للنظام المعينة بواسطة حالات الاستعمال. التكوين الأولي ينقى بإدخال عناصر جديدة الية للخصائص غير العملية.

إسهامنا يسكن في رد كل القرارات الخاصة بالهيكل واضحة، لاسيما التي تخص الأخذ

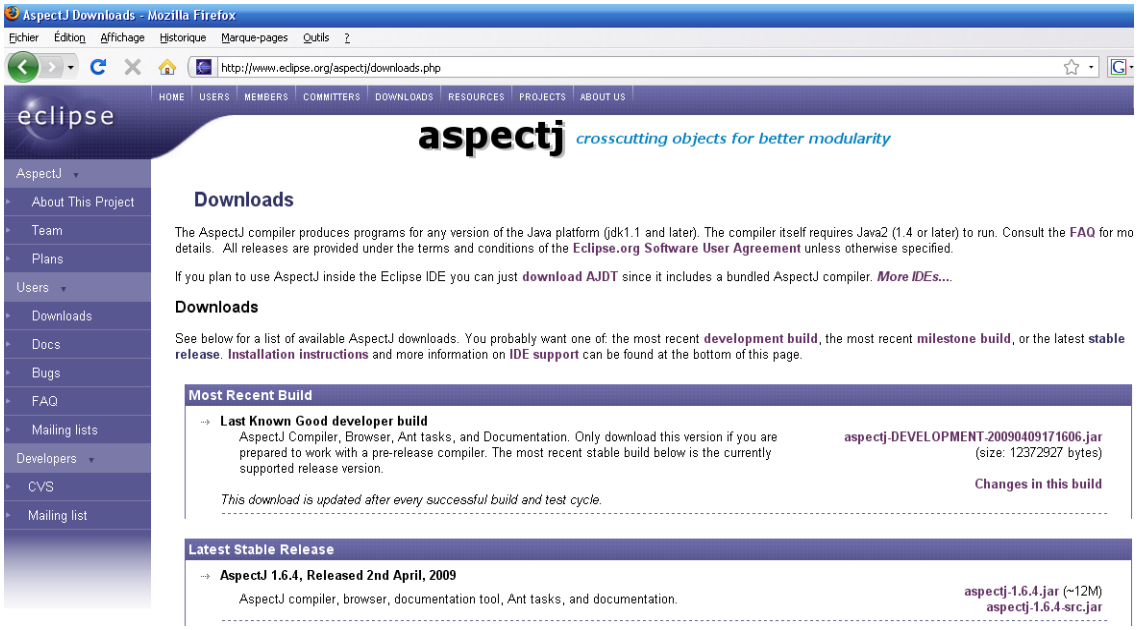
ANNEXES

Installation du kit de développement (AJDK)

Le compilateur aspectJ compile des fichiers .java . De plus il compile des fichiers .aj : les aspects. Exemple : Security.aj, Transactions.aj, Logs.aj...

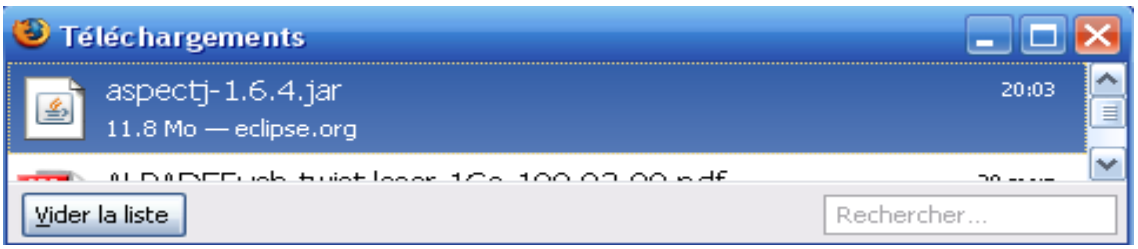
Téléchargement

- Allez sur le site de téléchargement de téléchargement dûaspectJ : http://www.eclipse.org/aspectj/downloads.php

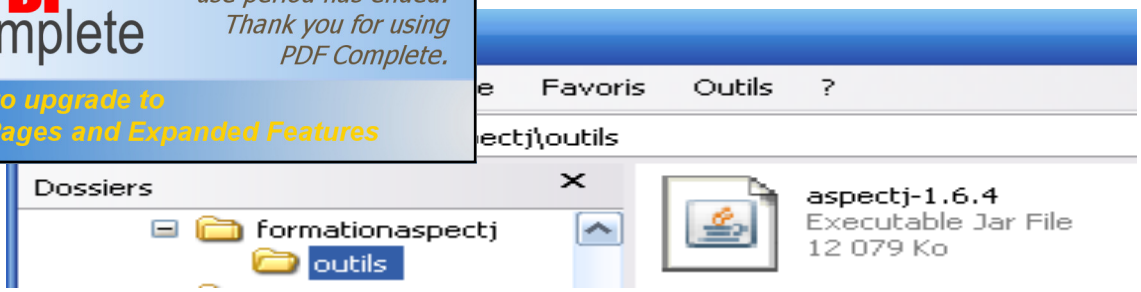


- Dans la section "Latest stable release" Cliquer sur le jar correspondant à la dernière version stable disponible (ici aspectj-1.6.4.jar). Vous obtenez un écran comme celui-ci. Une liste de sites miroirs apparaît.

- Choisir le site miroir proposé par défaut (ou choisissez celui que vous souhaitez). Le téléchargement démarre.



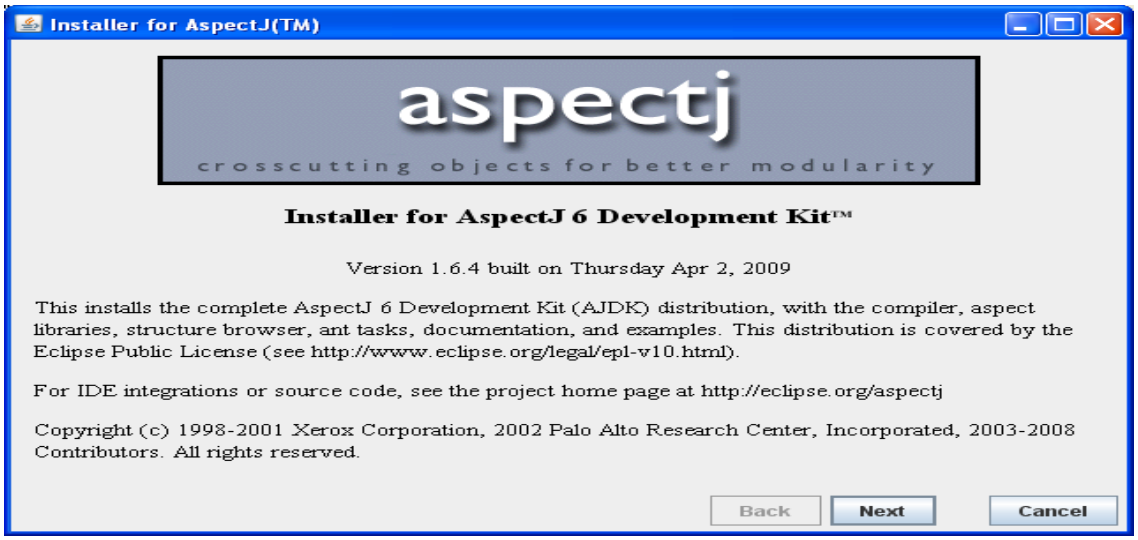
Une fois le téléchargement terminé, déposez le jar aspectj dans un répertoire, par exemple : c :\formationaspectj\outils .



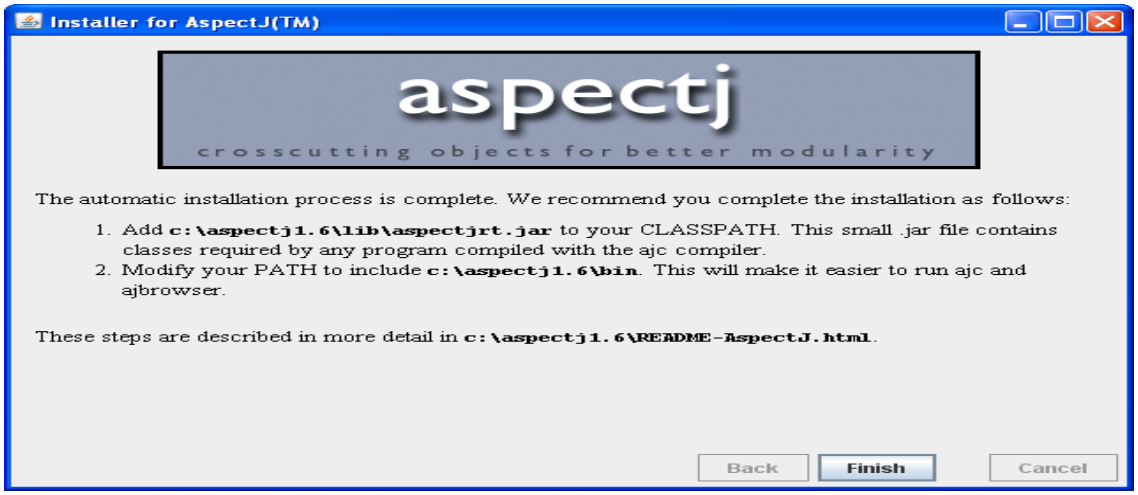
Reste désormais à installer...

Installation

- Double cliquer sur le jar aspectj : Lôassistant dôinstallation du kit de développement aspectJ démarre.



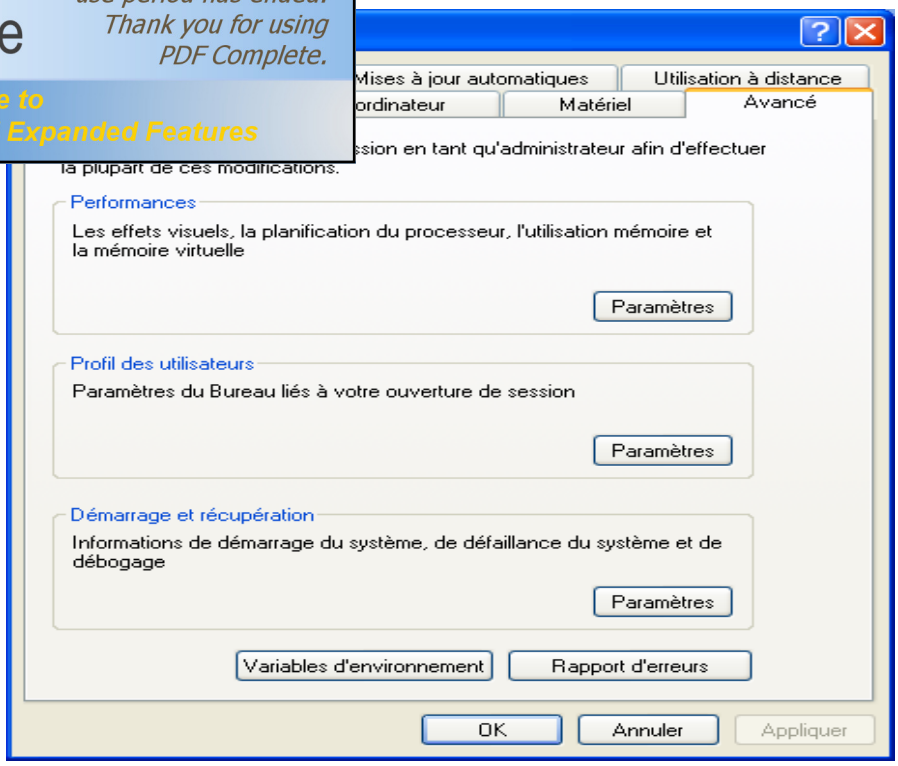
- Suivre les instructions de lôinstallation jusquôà la fin comme ci-dessous



- A la fin de lôinstallation, lôinstalleur vous informe des 2 actions manuelles à réaliser : Création ou mise à jour des variables dôenvironnement CLASSPATH et PATH.

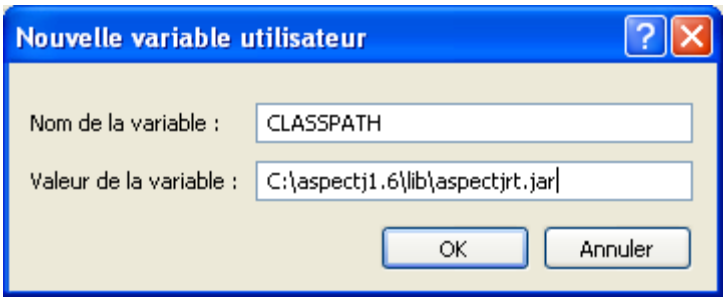
Variable d'environnement CLASSPATH :

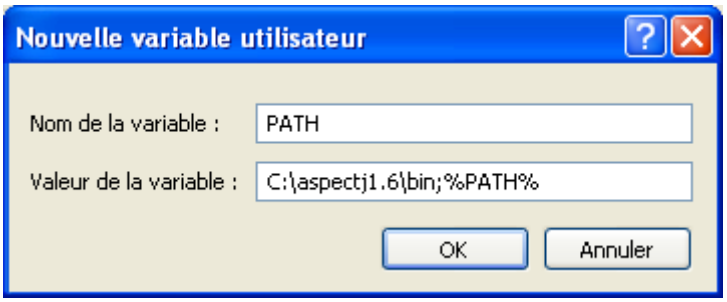
- Allller au Démarrer/panneau de configuration/système/Onglet ôavancéô



- Cliquer sur le bouton «variable d'environnement». Vous obtenez la fenêtre des variables d'environnement, contenant une partir «user» et une partie «système».

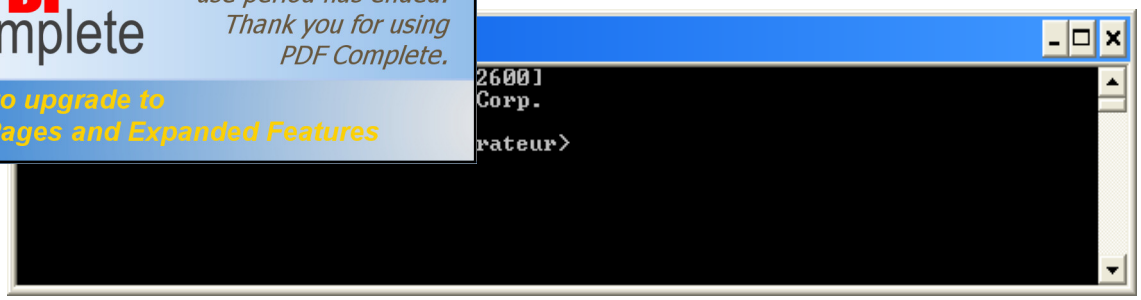
- Dans la partie «utilisateur», cliquez sur «nouveau». Ajoutez une variable **CLASSPATH** puis une variable **PATH** comme indiqué ci-dessous.





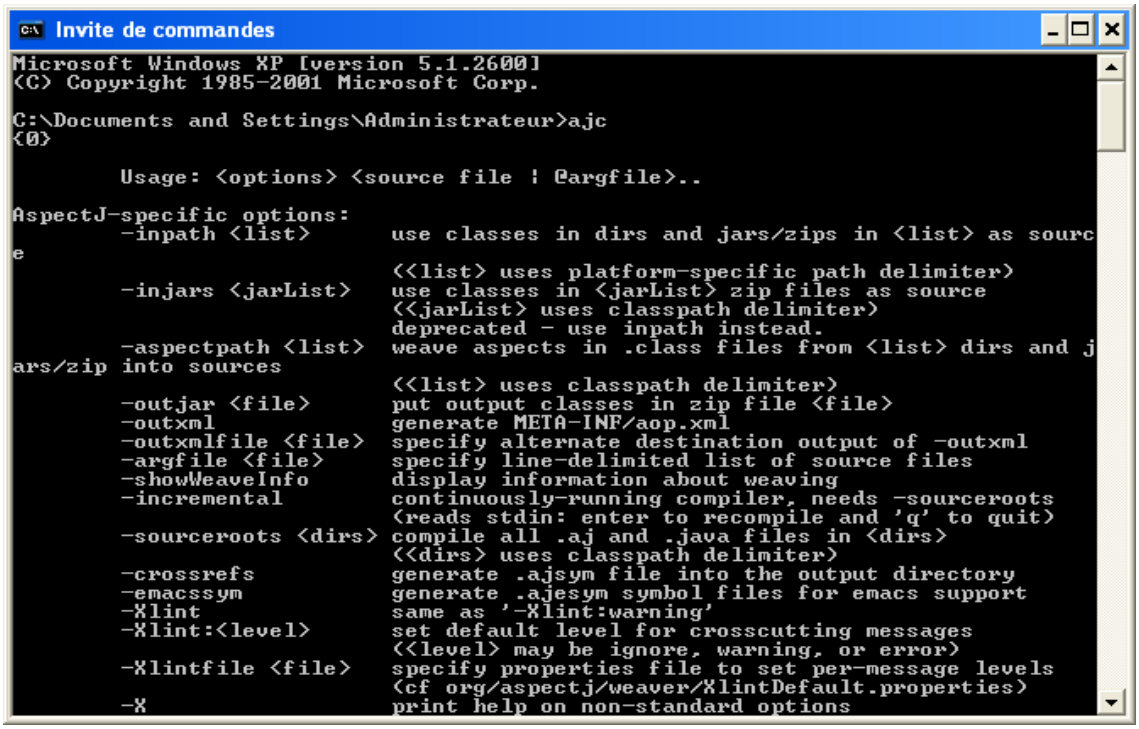
Test de l'installation

- Lancer un «Invite de commandes» MS-DOS (sous Windows xp : Démarrer/programmes/Accessoires/Invite de commandes)



- Lancer la commande : ajc

Nous appelons là le compilateur AspectJ (ajc : **a**spect**j** **c**ompiler).Les lignes suivantes apparaissent :



Lœenvironnement de développement AspectJ est désormais correctement installé sur notre ordinateur et est prêt à lœutiliser.

Les propriétés non fonctionnelles

Caractéristiques selon la référence [53] :

Capacité fonctionnelle (functionality) *: ensemble d'attributs portant sur l'existence d'un ensemble de fonctions et leurs propriétés données. Les fonctions sont celles qui satisfont aux besoins exprimés ou implicites.*

- **Aptitude (Suitability) :** Attributs de logiciel portant sur la présence et l'adéquation d'une série de fonctions pour des tâches données.
- **Exactitude (Accuracy) :** Attributs de logiciel portant sur la fourniture de résultats ou d'effets justes ou convenus.
- **Interopérabilité (Interoperability) :** Attributs de logiciel portant sur sa capacité à interagir avec des systèmes donnés.
- **Conformité réglementaire (Compliance) :** Attributs du logiciel selon lesquels il respecte l'application des normes, des conventions, des réglementations de droit ou des prescriptions similaires.
- **Sécurité :** Attributs du logiciel portant sur son aptitude à empêcher tout accès non autorisé (accidentel ou délibéré) aux programmes et aux données (comme nous avons vu avec notre exemple logging et vérification de solde).

Fiabilité (Reliability) : *ensemble d'attributs portant sur l'aptitude du logiciel à maintenir son niveau de service dans des conditions précises et pendant une période déterminée.*

- **Maturité (Maturity) :** attributs du logiciel portant sur la fréquence des défaillances dues à des défauts du logiciel.
- **Tolérance aux fautes (Fault tolerance) :** Attributs du logiciel portant sur son aptitude à maintenir un niveau de service donné en cas de défaut du logiciel ou de violation de son interface.
- **Possibilité de récupération (Recoverability) :** Attributs du logiciel portant sur ses capacités de rétablir son niveau de service et à restaurer les informations directement affectées en cas de défaillance, et sur le temps et l'effort nécessaires pour le faire.

Facilité d'utilisation (Usability) : *ensemble d'attributs portant sur l'effort nécessaire pour l'utilisation et sur l'évaluation individuelle de cette utilisation par un ensemble défini ou implicite d'utilisateurs.*

- **Facilité de compréhension (Understandability) :** attributs du logiciel portant sur l'effort que doit faire l'utilisateur pour reconnaître la logique et sa mise en oeuvre.
- **Facilité d'apprentissage (Learnability) :** attributs du logiciel portant sur l'effort que doit faire l'utilisateur pour apprendre son application (par exemple, maîtrise de l'exploitation des entrées et des sorties).
- **Facilité d'exploitation (Operability) :** attributs du logiciel portant sur l'effort que doit faire l'utilisateur pour exploiter et contrôler son exploitation.

ttributs portant sur le rapport existant entre le niveau

ressources utilisées, dans des conditions déterminées.

- **Comportement vis-à-vis du temps (Time behavior) :** attributs du logiciel portant sur les temps de réponse et de traitement ainsi que sur les débits lors de l'exécution de sa fonction.
- **Comportement vis-à-vis des ressources (Ressource behavior) :** attributs du logiciel portant sur la quantité de ressources utilisées et sur la durée de leur utilisation lorsqu'il exécute sa fonction.

Bibliographie

[1] Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*, Second Edition, Addison-Wesley, 1999.

[2] Sommerville Ian, *Software Engineering*, Sixth Edition, Addison-Wesley, 2001.

[3] Ghezzi Carlo, Jazayeri Mehdi, Mandrioli Dino, *Fundamentals of Software Engineering,* Second edition. Prentice Hall, 2003.

[4] Larman Craig, *Applying UML and Patterns*, Second edition. Prentice Hall, 2002.

[5] Wiegers, Karl, *Software Requirements. Second edition,* Microsoft Press, 2003.

[6] Perry William, *Effective Methods for Software Testing*, Addison-Wesley, 2000.

[7] ISO, ISO/IEC 9126-1 *Software engineering – Product quality : Quality model*, 2001.

[8] Jackson Michael, Problem Frames, Addison-Wesley, 2001.

[9] Ventimiglia Bernardo, Louis Martin, *A Case against de Use Cases in the Classroom ?,* http://www.trempet.uqam.ca/trempet/membres/Maffezzini/Articles/ArticlesGL/AgainstUsesCasesInTheClassroom.pdf, 1995.

[10] Roques, P. (2006a). Uml2 - modéliser une application web. Eyrolles.

[11] Roques, P. (2006b). Uml2 par la pratique (étude de cas et exercices corrigés) (5 ed.). Eyrolles.

[12] Rumbaugh, J., Jacobson, I. & Booch, G. (2004). Uml 2.0 guide de référence. CampusPress.

[13] Roques, P. (2002). Uml - modéliser un site e-commerce. Eyrolles.

[14] Renaud Pawlak, J.P. Retaillé, and Lionel Seinturier. Programmation orientée aspect pour Java/J2EE. Eyrolles, 2004. ISBN : 2-212-11408-7.

[15] Ivar Jacobson , Pan-Wei NG : Aspect-Oriented Software Development with Use Cases 2004

[16]. OMG Document ad/99-04-07 April 1999. ñWhite Paper on the Profile Mechanismò

[17] JosWarmer and Anneke Kleppe. The Object Constraint Language. Addison-Wesley, 1998.

[18] Ramnivas Laddad. Aspectj in Action Practical Aspect-Oriented Programming,Manninig,Greenwich,CT,2003.

[19] Laurent AUDIBERT : UML 2 Edition 2007/2008.

èmes : De l'approche fonctionnelle à l'approche objet.

[21] M. Basch and A. Sanchez *''Incorporating aspects into the UML''*. 3rd International Workshop on Aspect-Oriented Modeling (in AOSD 2003), Boston, USA, (2003).

[22] Muhammad Ali Babar and Barbara Kitchenham. Assessment of a framework for comparing software architecture analysis methods. In EASE, Keele University, UK, April 2007. 11th International Conference on Evaluation and Assessment in Software Engineering (EASE).

[23] Alistair Cockburn, « Rédiger des cas dûtilisation efficaces », Eyrolles, 2001.

[24] O. Hachani, Thèse de Doctorat, *"Patrons de conception a base d'aspects pour l'ingénierie des systèmes d'information par réutilisation*ñ, Université de Joseph Fourier Grenoble 1, (2006).

[25] Y. Han, G. Kniesel and A. B. Cremers *''A meta model and modelling notation for AspectJ''*. In the 5th Aspect-Oriented Modeling Workshop in conjunction with UML 2004, Lisbon, Portugal, (2004).

[26] N. Jayaratna. Normative information model based systems analysis and design (nimsad) : A frame work for understanding and evaluation methodologies. In Journal of applied analysis, pages 73_87, 1986.

{27] Sylvain SAUVAGE. Conception de systèmes multi-agents : un thésaurus de motifs orientés agent. PhD thesis, l'Université de Caen Basse Normandie, France, octobre 2003.

[28] M. Shaw and D. Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, Computer Science Today : Recent Trends and Developments, volume 1000 of Lecture Notes in Computer Science, pages 307_323. Springer-Verlag, 1995.

[29] D. Stein, Thèse de Master *''An Aspect-Oriented Design Model Based on AspectJ and UML''*, University of Essen, Germany, (2002).

[30] J. Suzuki and Y. Yamamoto *" Extending UML with Aspects: Aspect Support in the Design Phase".* ECOOPð99 Workshop on Aspect-Oriented Programming, (1999).

[31] Danny Weyns. An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems. PhD thesis, Katholieke Universiteit Leuven, Arenbergkasteel, Leuven, Belgium, 2006.

[32] A. A. Zakaria, H. Hosny and A. Zeid *'' A UML Extension for Modeling Aspect-Oriented Systems''*. 2nd International Workshop on Aspect-Oriented Modeling with UML. (2002).

[33] M.J. Lions, D. Simoneau, G. Pitette and I. Moussa ' Extending OpenTool/UML Using Metamodeling: An Aspect-Oriented Programming Case Study. 2nd International Workshop on Aspect-Oriented Modeling with UML (UML 2002), September 2002.

ames Rumbaugh, The Unified Software Development

ert, and F. Buschmann. Pattern-Oriented Software
nd Networked Objects, volume 2. Wiley, Chichester, 2000.

[36] D. Parnas. ((On the criteria to be used in decomposing systems in modules)). Communication on the ACM, 15(12):1053-1058, 1972.

[37] C. Lopes et W. Hursch. ((Separation of Concerns)). Rapport technique, College of Computer Science, Northeastern University, Boston, MA, Etats-Unis, Février 1995.

[38] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, et J. Irwin. « Aspect-Oriented Programming ». Dans Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), volume 1241 de Lecture Notes in Computer Science, pages 220-242. Springer, juin 1997. « »

[39] Karl J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. Addison-Wesley,, 2000. ISBN 0-534-94602-X.

[40] M. Aksit, K.Wakita, J. Bosch, et L. Bergmans. « Abstracting Object Interactions Using Composition Filters». volume 791, pages 152_184, 1994.

[41] K. Lieberherr, D. Lorenz, et M. Mezini. ((Programming with aspectual components)). Rapport Technique Technical Report NU-CCS-99-01, Northeastern University's College of Computer Science, apr 1999.

[42] M. Mezini et K. Lieberherr. « Adaptative plug-and-play components for evolutionary software development ». SIGPLAN Notices, 33:96{116, 1998. In Proceedings of OOPSLA' 98, ACM Press.

[43] M. Mezini, L. Seiter, et K. Lieberherr. « Component integration with pluggable composite adapters », chapitre Software Architectures and Component Technology: The State of the Art in Research and Practice. In L. Bergmans and M. Aksit, kluwer academic publishers edition, 2001.

[44] W. Harrison et H. Ossher. « Subject-oriented programming (A critique of pure objects) ». Dans Proceedings of OOPSLA'93, volume 28 of SIGPLAN Notices, pages 411-428, octobre 1993.

[45] H. Ossher, K. Kaplan, W. Harrison, A. Matz, et V. Kruskal. « Subject-Oriented Composition Rules ». Dans Proceedings of OOPSLA'95, volume 30 de Sigplan Notices, pages 235-250. ACM Press, 1995.

[46] H. Ossher et P. Tarr. ((Multi-dimensional separation of concerns and the hyperspace approach)), chapitre Software Architectures and Component Technology: The State of the Art in Research and Practice. In L. Bergmans and M. Aksit, kluwer academic publishers edition, 2001.

ison, et S. Sutton. « N degrees of separation: s ». Dans Proceedings of the International Conference ges 107-119, 1999).

[48] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, et W. Griswold. « An Overview of AspectJ ». 2001. AspectJ white paper submitted to the the European Conference on Object-Oriented Programming (ECOOP'01).

[49] Shyam R. Chidamber et Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6): 476ï 493, Juin 1994.

[50] Norman Fenton et Shari Lawrence Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2*ème* édition, 1996. isbn : 1-85032-275-9.

[51] Gregor Kiczales, Tzilla Elrad, Mehmet Ak¸sit, Karl Lieberherr et Harold Ossher. Discussing aspects of AOP. *Comm. ACM*, 44(10): 33ï 38, Octobre 2001.

[52] Object Management Group. Unified Modeling Language (UML), version 2.0, Specifications. *http: // www. uml. org/* , 0 :0, 2005.

[53] Qualité logicielle. Site web (http: //lil.univ-littoral.fr/ oumoum-sack/qualité).