UNIVERSITÉ DU QUÉBEC

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME PARTIELLE EXIGENCE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ERIC DALLAIRE

INTÉGRATION D'UN MODÈLE D'ADAPTATION DYNAMIQUE ET DE SÉCURITÉ D'OBJETS AVEC VUE

HIVER 2010

Sommaire

La programmation orientée objet est une solution idéale pour une multitude de problèmes depuis plus de deux décennies. Des langages tels que le C++ et le Java ont largement contribué à l'expansion. Ce succès repose en partie sur une raison toute simple : il est plus naturel d'utiliser des objets qui se rapprochent de la réalité « humaine » que de travailler avec des fonctions séquentielles généralement associées à des processus informatiques. Il est relativement aisé d'identifier les caractéristiques et les comportements d'un objet et toute bonne analyse permet de gérer adéquatement les interrelations entre les objets dans un système.

Toutefois, on limite parfois notre réflexion à un stade d'abstraction insuffisant pour aller chercher un maximum d'avantages des objets. Les objets sont correctement développés, dans un système répondant à un nombre fini d'évènements, mais on néglige le fait que ce même système est en contact avec un élément chaotique et imprévisible, l'humain.

Prenons exemples des comportements dynamiques entre les objets, des divers patrons de conception déjà avancés, mais surtout de la question des comportements de ces objets (ayant des comportements figés) face aux besoins réels d'un utilisateur en constant changement. Après un certain temps, un système informatique peut évoluer dans ses comportements et ses rôles. De nouvelles fonctionnalités sont, à travers son cycle de vie, ajoutées, supprimées ou deviennent tout simplement non disponibles. Aussi, chacun des acteurs du système, autant les clients humains que les systèmes informatiques externes, a des besoins variables qui peuvent exiger un changement de vocation du système utilisé. Ce dernier doit être évolutif pour y répondre. À une échelle d'entreprise, on peut dire que plusieurs clients observent et utilisent des aspects distincts du même objet. Que faire si cet objet ne peut satisfaire aux différents intérêts du client?

La programmation par vue a pour mission de répondre à cette situation. Elle permet à différents usagers de percevoir et utiliser différents sous-ensembles des fonctionnalités présentes au sein de l'objet a un moment donné. Le client a un besoin un jour, la programmation par vue lui propose une interface adaptée, son besoin change le lendemain, les vues évoluent en conséquence.

La programmation par vue existe depuis quelques années, mais certains questionnements étaient demeurés sans réponse. Le sujet du présent mémoire sera de fournir quelques-unes des réponses à une série de problématiques entourant principalement i) la sécurisation de l'objet partagé selon les privilèges de chaque programme client, ii) la transparence et la gestion de changement de comportements de l'objet en question, et iii) l'intégration de mécanismes de partage des ressources.

Nous avons étendu le modèle de sécurité de Java afin de pouvoir gérer l'évolution des comportements. Java propose un modèle d'utilisation simple demandant une configuration minimale. Notre solution vise à garder une approche le plus transparente possible pour le client et aussi pour la classe de base (le système) que l'on désire faire évoluer.

Notre recherche abordera la programmation par vue avec extension de sécurité à l'aide d'énoncés de problématiques. Le développement de notre outil sera accompli via une solution conceptuelle en UML et une implémentation en Java.

Remerciements

La rédaction de ce mémoire a été une expérience laborieuse accomplie en parallèle à ma vie familiale et à mes activités professionnelles collégiales et universitaires. Il est difficile de garder la flamme lorsque l'étude passe souvent en troisième plan, très tard le soir ou les matins de fin de semaine. Toutefois, j'ai réussi et j'en éprouve une grande fierté.

Depuis le début de notre association, mon directeur, Hamid, a été tel que j'osais l'imaginer. Toujours disponible pour répondre à mes vagues de questions par courriel ou pour des rencontres improvisées. Un excellent guide lors de mes égarements et très compréhensif lorsque je travaillais jour et nuit sans pouvoir trouver une seconde pour avancer mon projet. En écrivant ces lignes, je me rends compte que Hamid a été capable de me pousser à avancer, sans pression ni sanction, tout en démontrant un intérêt soutenu à mon évolution. Pour ces raisons et pour plusieurs autres, je lui suis vraiment reconnaissant.

Il m'est impossible de passer sous silence l'énorme contribution de ma femme Isabelle (merci pour les images ;-), de mes enfants Samuel, Catherine, Elysabeth, Eléonore et Victor ainsi que mes parents, mon frère et mes beaux-parents. Je voulais terminer ma maîtrise à tout prix et ils m'ont appuyé de tout leur cœur, sans défaillir. Aucun mot ne pourra jamais remplacer le temps perdu avec eux.

Finalement, un dernier remerciement s'adresse au département d'informatique et de mathématique de l'UQAC. Plusieurs personnes, à travers des discussions et des partages d'expérience, m'ont offert encouragements, support et conseils à des moments critiques et m'ont ainsi permis de persévérer dans l'échec et d'oser dans la réussite.

Tables des matières

Listes de figures

Listes de tableaux

1

INTRODUCTION

L'utilisation des systèmes informatiques a pris une ampleur indescriptible depuis les dernières années. Tous les domaines, aussi fermés étaient-ils à la technologie, ont dû faire face à l'inévitable intégration. Depuis plusieurs décennies, des systèmes allant des plus simples aux très complexes se sont développés au gré des langages et des solutions. Certains se sont révélés des échecs, mais d'autres, même à l'heure actuelle, fonctionnent toujours.

Avec l'évolution des besoins, de bons systèmes informatiques deviennent désuets ou sur le point de le devenir. Par exemple, un programme de gestion du système de paie écrit en Cobol peut-il encore aujourd'hui être interconnecté avec tous les autres systèmes modernes? Peut-il encore répondre à nos requêtes d'information? On pourrait être tenté de répondre par l'affirmative, mais il ne faut pas oublier le facteur financier inquiétant qui hante toute modification.

La programmation orientée objet, notamment avec l'avènement du C++ [Stroustrup, 07-1], se voulait la solution à tous les problèmes. Son but est d'organiser le code et de concevoir le programme tout en diminuant le coût de maintenance. Une classe est en fait une représentation d'une idée, d'un concept et il est possible de relier ces concepts

ensemble afin de former des relations consistantes [Stroustrup, 07-2]. En effet, un système développé avec une approche plus conceptuelle que programmaticale, s'éloigne de la machine et se rapproche de la réalité humaine, ce qui le rend plus adaptatif aux variations des besoins.

Traditionnellement, dans les techniques de programmation orientée objet (POO), la classe est considérée comme l'entité de base de toute conception. Elle représente l'élément unitaire et indivisible qui permet de construire la structure de tout programme. Une classe fournit à l'utilisateur une interface complète comprenant l'ensemble des actions possibles (ses méthodes) et des caractéristiques disponibles (ses attributs). Toutefois, le système orienté objet reste figé, la classe étant compilée telle que conçue.

En réalité, la classe fait partie d'un ensemble de classes reliées entre elles à l'intérieur d'un système informatique. Dépendamment de l'implémentation, son rôle peut varier selon l'utilisation qu'on en fait ou encore selon le moment de l'utilisation (l'état de l'objet à un moment particulier). Des interfaces pourraient être ajoutées afin de limiter les accès à l'objet, car tout utilisateur n'a pas à être en contrôle total de toutes ces fonctionnalités. Toutefois, l'utilisation des interfaces est *statique et une fois compilée on ne peut plus la modifier*.

Dans l'exemple de la figure 1.1, chaque méthode de l'objet est atteignable grâce à des interfaces distinctes soit I_1 pour atteindre f(x), I_2 pour g(x) et I_3 pour h(x). Pour accéder à f(x), g(x) et h(x), l'utilisateur devra utiliser ces trois interfaces et les ajouter directement dans son code pour effectuer la compilation. Le nombre de possibilités augmente rapidement.

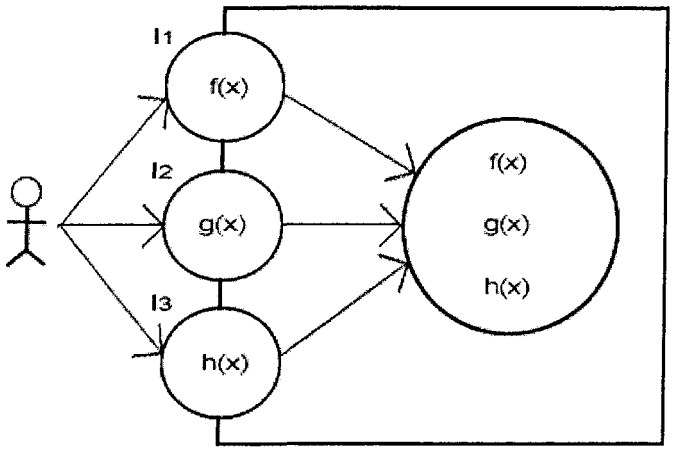

Figure 1.1 : Un client en relation avec trois interfaces

Les interfaces découpent assez bien les classes, mais leur rôle reste limité. Un utilisateur questionne un objet à travers une interface fixe qui n'est pas obligatoirement basée sur une approche fonctionnelle. En fait, les interfaces actuelles en programmation s'apparentent, par la restriction des accès, aux concepts de vue que l'on retrouve dans le domaine des bases de données [Gardarin, 01]. Par exemple, un objet InfoEtudiant, contenant toutes les informations possibles sur un profil étudiant, pourrait être questionné en utilisant une série d'accesseurs et de méthodes. Il pourrait également hériter d'une interface *Runnable* (qui le définit comme un thread) ce qui n'apporte, fonctionnellement, pas grand-chose à l'utilisateur.

Citons un problème très courant des systèmes orientés objet. On veut partager l'objet InfoEtudiant afin de permettre son utilisation, mais on ne veut pas offrir sur un « plateau d'argent » toutes ses possibilités. Il existe des mécanismes de protection de base qui permettent de protéger l'accès aux membres (par exemple l'encapsulation en Java) et aussi

les interfaces pour diviser les accès aux membres publics. Malheureusement, le problème reste entier, car le propriétaire de l'objet peut l'utiliser à sa guise tout simplement en changeant d'interface.

Pour continuer avec notre exemple, que faire si on désire encadrer les accès ou encore ajouter des fonctionnalités supplémentaires à notre classe? Il est naturellement possible de créer de nouvelles interfaces personnalisées aux besoins. Le problème c'est qu'on doit, à chaque nouveau besoin, modifier et recompiler l'objet, mais aussi ajuster la classe appelante ce qui ramène rapidement au problème soumis au début de cette introduction : il faut limiter les modifications à notre système.

Ajoutons trois interfaces à la classe InfoEtudiant pour répondre à un nouveau besoin fonctionnel : soit les interfaces Finance, Administration et Archive (voir figure 1.2). Supposons que l'interface Finance fournisse à l'utilisateur une série de fonctionnalités reliées au domaine d'affaire et que les interfaces Administration et Archive en fassent de même à leur façon. Supposons aussi que chacune possède des méthodes communes, mais ayant des réalisations différentes. La classe de base InfoEtudiant doit déjà être passablement modifiée pour répondre à ces nouvelles promesses de fonctionnalités. Que se passe-t-il maintenant si un utilisateur veut passer d'une interface à une autre? Ou encore, d'utiliser plus d'une interface à la fois? Nous aurions possiblement une dizaine de combinaisons à développer…

À travers son cycle de vie, un objet peut évoluer et doit changer de comportements [Mili et al., 99]. Il doit être en mesure de réagir aux besoins fonctionnels émis par son environnement tout en conservant son intégralité. Chaque besoin fonctionnel est en fait

une vue spécifique de notre objet, une voie de communication [Mili et al., 06]. On peut représenter *une vue comme un intermédiaire fonctionnel à un objet qui peut évoluer.* Si les besoins changent, on change de vues.

En réalité, un profil étudiant est à la fois un élément d'un bilan financier, un objet d'administration et un item d'inventaire. Des vues peuvent être ajoutées dynamiquement à l'objet de base (ObjAppEtudiant). La programmation par vue aborde cet aspect de la programmation objet qui sera traité en profondeur au chapitre 2.

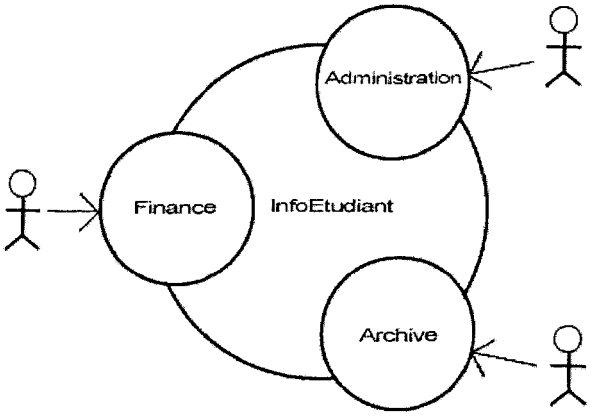

Figure 1.2 : Vision simplifiée de l'objet de base « InfoEtudiant »

Avant de poursuivre, il est important de noter la différence entre une interface de programmation traditionnelle (statique) et une vue. Une vue est une implémentation contextuelle d'un point de vue abstrait. Les interfaces peuvent être utilisées lors de l'implémentation de vues. Ce document traite particulièrement des vues de la programmation par vue.

1.1 Énoncé de la problématique

La programmation par vue étant déjà une approche finement développée, l'objectif de la présente recherche est de proposer une extension au modèle existant. Actuellement, les programmes clients doivent spécifier eux-mêmes leurs besoins des fonctionnalités (vues) et demander explicitement l'activation et la désactivation de ces vues auprès d'un objet donné [Mcheick, 06]. Or, les objets pourraient eux-mêmes s'auto adapter et activer/désactiver des vues en fonction des méthodes demandées sans intervention des programmes clients. Notre objectif est de:

- Développer un modèle conceptuel de l'auto-adaptation des objets
- Implémenter ce modèle pour faire évoluer les fonctionnalités des objets
- Traiter le problème de sécurité. Il s'agit de trouver et d'implémenter un modèle de sécurité simple et léger pour préserver les droits d'accès de chaque programme client sur un objet partagé

1.2 Principe de solution

Pour permettre aux objets d'activer et de désactiver des vues sans intervention des programmes clients, et ainsi préserver les privilèges de chacun de ces programmes, nous proposons un *modèle d'adaptation dynamique et de sécurité d'objets avec vues.*

Afin d'arriver à une solution viable, nous proposons une solution en trois étapes. Tout d'abord, la première étape consiste à *rendre l'utilisation de notre objet transparent à l'utilisation pour le client.* Actuellement, le client doit activer et désactiver ses vues pour utiliser l'objet selon son besoin. Cette fonctionnalité sera complètement changée pour être

remplacée par une gestion du côté de l'objet. Pour en arriver à permettre une relation entre le client et l'objet, le client devra être identifiable hors de tout doute. Tout appel d'un programme client sera sujet à une politique de sécurité qui définit l'ensemble des permissions disponibles pour un programme client particulier. Ce modèle fera usage du gestionnaire de sécurité de Java [SUN, 07].

La seconde étape sera de *reconnaître le client dans l'objet*, ce dernier ayant été préalablement défini comme identifiable. Il faut que l'objet sache de qui provient l'appel afin de mettre les bonnes vues à la disposition de l'appelant.

La dernière étape repose complètement sur la *gestion des accès à l'aide de permissions personnalisées* utilisant le système de permissions de Java [SUN, 07]. Chaque utilisateur aura accès à une certaine quantité de vues qui seront rendues disponibles pour son utilisation.

On propose donc l'ajout d'un modèle de sécurité des vues [Mcheick et Dallaire, 08] qui se superposera discrètement, et de la manière la plus transparente possible, au dessus du modèle de programmation par vue actuel.

1.3 Contribution

La présente recherche propose de réaliser l'analyse de la programmation par vue courante et de contribuer à son évolution à travers diverses activités :

- Insérer, tout en respectant les principes fondamentaux de l'approche, une gestion dynamique des vues qui se voudra sécuritaire, légère et efficace.

- Expérimenter certaines propositions que nous avançons pour le développement ou la modification de certaines sections de la programmation par vue. Par exemple, le passage du contrôle des comportements de l'objet d'application de la main du client à celle de l'objet lui-même afin de le rendre plus autonome. Un autre exemple, nous tenterons de fournir une solution avec un minimum de lourdeur de programmation, des classes noyaux plus complexes offrant un maximum de possibilité de réutilisation.
- Proposer un modèle théorique de sécurité des vues basé sur JAAS via UML
- Réaliser une implémentation finale en Java

1.4 Structure du mémoire

Ce travail de recherche est divisé en six chapitres. Le présent chapitre introduit la problématique générale du développement basée sur les besoins utilisateur ainsi qu'un court extrait des objectifs et solutions du modèle de sécurité des vues.

Le second chapitre aborde la programmation par vue afin de permettre la compréhension des divers éléments du domaine qui seront abordés dans les chapitres subséquents. Il y sera aussi fait mention des autres champs d'études reliés à notre sujet.

Le troisième chapitre présentera un survol du modèle de sécurité de Java. Le fonctionnement du modèle de permissions qui sera utilisé pour la solution y sera particulièrement approfondi.

Le chapitre quatre aborde en détail l'énoncé de la problématique. Basées sur les concepts de la programmation par vue et utilisant les particularités du modèle de sécurité de

Java abordée dans les chapitres précédents, diverses problématiques seront soulevées afin de faire ressortir ce qu'on cherche vraiment à apporter.

Le chapitre cinq contient l'approche proposée, côté conceptuel via des modèles UML et côté pratique à travers des exemples de code Java. Les résultats de l'implémentation seront finalement analysés dans le chapitre suivant.

Enfin, le mémoire se termine par la conclusion où on pourra retrouver les propositions finales découlant de nos réflexions, observations et expérimentations.

2

LA PROGRAMMATION PAR VUE

2.1 Introduction

Après un certain temps, un système informatique peut évoluer dans ses comportements et ses rôles. De nouvelles fonctionnalités sont, à travers son cycle de vie, ajoutées ou supprimées, activées ou désactivées [Mili et al., 2002] [Mcheick, 06].

Aussi, chacun des clients du système, autant les usagers humains que les systèmes informatiques externes, ont des besoins variables qui peuvent exiger un changement de vocation du système utilisé. Le système doit être évolutif pour y répondre. À plus petite échelle, on peut dire que plusieurs clients observent et utilisent des aspects distincts du même objet. Afin de permettre l'utilisation adéquate de l'objet, il convient d'insérer des mécanismes de gestion et de sécurité.

Une série de mécanismes de gestion existent déjà dans la plupart des langages objet (ex. : gestionnaire de sécurité de Java, polymorphisme). Beaucoup de leurs fonctionnalités techniques sont bien connues et bien documentées. Toutefois, il devrait être possible de pouvoir regarder l'objet de base sous un angle plus fonctionnel que technique. Prenons l'exemple d'un dossier étudiant qui serait représenté par une classe InfoEtudiant.

Dynamiquement, on devrait être en mesure de décider si les informations de l'étudiant existeraient en tant qu'ensemble de données nécessaire à la création d'un rapport ou encore à la génération d'un bordereau de paie. De plus, à un moment ultérieur, la classe InfoEtudiant pourrait être utilisée pour créer et tenir à jour un profil d'apprentissage dynamique dans un site en ligne. Dans cette mise en situation, l'objet de base InfoEtudiant ne devrait pas subir de modifications à chaque apparition d'un nouveau besoin. Il serait même préférable qu'il ne soit pas conscient de toutes ces modifications de besoin. On arrive rapidement à la limite des interfaces conventionnelles dont l'utilisation est fixée à la compilation de la classe.

2.1.1 Tour d'horizon

Dans le but de faire évoluer le rôle de l'objet et de permettre son adaptation, diverses solutions ont été apportées au fil des ans. Parmi les approches existantes, nous distinguons notamment la programmation par sujet (Subject-Oriented Programming SOP) [CSS, 01], la programmation par aspect (Aspect Oriented Programming AOP) [DotNetGuru, 03] et la programmation par vue (View oriented Programming VOP) [Mili et al., 99].

La première et la plus ancienne, la programmation par sujet, permet de fusionner des applications ayant une interface semblable et offrant une implémentation différente de la même entité. Par exemple, deux applications P1 et P2 ayant chacune une interface Employe. La programmation par sujet propose de les combiner afin d'obtenir une interface unique (Employe_P1 $\oplus$ Employe_P2 $\rightarrow$ Employe) [Harrison et Ossher, 1993]. Cette approche permet d'élargir et de compléter les rôles de chaque objet.

Avec la programmation par aspect, la séparation des préoccupations peut être utilisée comme une solution permettant d'identifier, d'encapsuler et de manipuler les parties d'un logiciel qui sont pertinentes pour un domaine particulier. Ces dernières sont ensuite composées grâce à des points de jointure selon l'approche utilisée. La séparation entre les préoccupations (rôles) qu'un objet peut jouer facilite la maintenance, la compréhension, la réutilisation des applications et permet la modularité [Kiczales et al., 97]. Par exemple, on pourrait identifier une préoccupation reliée à la sécurité, définir un code de gestion en Java et l'intégrer à des points de jointure prédéfinis à l'aide d'outils (*ex : AspectJ* [Eclipse, 03])

La programmation par vue est l'approche qui sera utilisée pour cette étude. On suppose qu'une entité peut jouer différents rôles fonctionnels à différents moments de sa vie [Mcheick, 06]. La prochaine section aborde en détail la programmation par vue.

Allant de pair avec la gestion des comportements, il faut aussi considérer la gestion des accès. Plusieurs approches existent mais deux ont particulièrement retenues notre attention : RBAC et JAAS. La gestion des accès par rôle (RBAC), reposant à la base sur une solution développée par le département de la justice américain [NIST, 09], a été habilement implémentée dans un contexte client-serveur reposant sur l'infrastructure CORBA [Fink et al., 03]. Le principe peut être résumé simplement. Un client est en relation avec un serveur (ex : Tomcat) offrant des accès à des services Web. Le serveur permet l'authentification des usagers, mais le processus d'autorisation se limite à permettre d'utiliser ou non un service Web. À l'aide d'un langage basé sur le XML (View Policy Language (*VPL*)) et d'outils de configuration, la solution permet de prendre le relai et de gérer les accès au niveau des méthodes. Ceci étant, chaque rôle se voit attribuer des droits

d'accès à certaines méthodes. Ce modèle est notamment implémenté dans le logiciel Microsoft Exchange Server 2010 où il permet d'effectuer la gestion des permissions [Microsoft, 09]. Notre modèle de sécurité ne suivra toutefois pas cette approche, quoique très intéressante, car elle est trop lourde (service Web, Corba ou EJB) et contraignante (client-serveur) pour une communication simple entre deux objets (Objet de base et objet d'application)

Le modèle de sécurité de notre solution fera usage de JAAS (Java Authentication and Authorization Service), une librairie de développement gratuite standardisée par Sun Microsystems. Le chapitre 3 y est entièrement consacré.

2.2 Description

La programmation par vue est une alternative à la programmation par aspect. On cherche à voir le système, et la classe à moindre échelle, comme une entité qui pourrait jouer différents rôles fonctionnels à différents moments de sa vie. On ne tente pas de s'insérer à travers les entités à l'aide de coupes transversales plutôt techniques, ce qui est déjà réalisé par la programmation par aspect. Une approche fonctionnelle avec un certain degré de recul est privilégiée. On doit définir une méthode d'utilisation de l'objet, sans intervention dans le code ni dans le processus déjà en place. Continuons avec l'exemple de la classe InfoEtudiant qui existe, compilée, dans un système. L'ajout d'un module d'archivage ou d'administration pourrait réclamer de nouvelles fonctionnalités à la classe. Par exemple, InfoEtudiant pourrait être initialement instanciée comme un objet d'administration et pourrait, à tout moment, devenir un objet archivable avec les méthodes

spécifiques appropriées. Pour réussir ce tour de force, on doit encapsuler l'utilisation de la classe InfoEtudiant afin de permettre son emploi sous une autre facette tout en gardant son intégralité. Nous fournirons donc à l'utilisateur de ce nouvel objet une interface adaptée à ses besoins fonctionnels indépendante ou en complément à l'objet lui-même. La programmation par vue a pour objectif de supporter cette évolution de comportement des objets dans le langage hôte.

2.3 Principes de base

On suppose que chaque objet du domaine d'application prend en charge un ensemble de fonctionnalités de base offertes à tous les programmes utilisateur et d'autres fonctionnalités qui sont spécifiques à certains autres usagers de l'objet. Cette supposition généralise les interfaces de visibilité (privées, protégées, publiques, *package*) dans les langages de programmation objet tels que Java ou C#. Les interfaces peuvent aussi correspondre à des domaines fonctionnels différents ayant leurs propres attributs et méthodes non offerts par l'objet de base. En fait, les interfaces, en ce sens, procurent un rôle particulier à un objet de base plutôt qu'une simple restriction d'accès.

La programmation par vue définit les principes de base suivants qui devront conserver leur véracité tout au cours de notre étude :

- Un objet peut changer de comportement durant l'exécution, en acquérant et perdant des fonctionnalités durant l'exécution (dynamisme)
- On doit pouvoir utiliser l'objet de façon transparente indépendamment du fait qu'il offre présentement la fonctionnalité ou non (transparence)

- On doit être capable d'accéder à plusieurs vues simultanément (simultanéité)
- Chaque comportement/vue doit avoir ses propres données qui doivent être préservées entre les activations (permanence)
- Lorsqu'un comportement est pris en charge par plusieurs vues, toutes les versions de ce comportement sont invoquées selon la composition précisée au développement (structure)
- La programmation par vue repose sur le concept de points de vue qui sont des patrons génériques. Ces patrons sont applicables sur un domaine d'application donné pour obtenir une vue (abstraction)

Reprenons l'exemple de notre classe InfoEtudiant représentant des informations reliées au profil étudiant. Supposons que l'on veuille vérifier si l'étudiant est endetté, mais que cette fonctionnalité n'existe pas dans l'objet InfoEtudiant. On prend la liberté d'ajouter une méthode enDette(...) afin de questionner adéquatement l'objet de base. Cette méthode ne sera pas définie dans l'objet lui-même, mais ajoutée à travers l'implémentation des vues. Avec cette nouvelle utilisation, on ajoute une couche d'abstraction. De ce fait, on n'aura peut-être plus accès, par choix de l'implémentation, à l'ensemble des méthodes de la classe InfoEtudiant. Au besoin, cette dernière pourra toujours être utilisée sous sa forme originale.

La programmation par vue prévoit également qu'un client puisse utiliser, à sa guise, la classe InfoEtudiant. Il pourra être un client administratif pour un instant et devenir un outil de génération de rapport financier ou même encore les deux à la fois. La section 2.4.5 traite des méthodes d'accès simultanés à l'objet

2.4 Éléments de la programmation par vue

La programmation par vue introduit de nouveaux termes et fonctionnements qui complètent ceux déjà connus et exploités à travers la programmation orientée objet classique. Voici la liste des principaux éléments :

- Vues et points de vue
- Objet de base et objet d'application
- Gestion des vues
- La délégation des appels
- La composition des vues

La présentation de notre approche se fera avec l'introduction de chacun de ces concepts.

2.4.1 Vue et points de vue

Supposons un objet compilé dans sa structure statique. On peut aisément en énumérer les interfaces d'utilisation, car on sait que toute classe possède un nombre fini de membres publics. Par exemple, en Java, une classe InfoEtudiant hérite, par défaut, de la classe Object (figure 2.1).

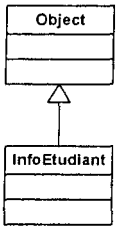

Figure 2.1 : Relation d'héritage de InfoEtudiant

Deux instanciations possibles de InfoEtudiant

```
InfoEtudiant infoEtudiant1 = new InfoEtudiant();
Object infoEtudiant2 = new InfoEtudiant();
```

En utilisant l'instance infoEtudiant1, on peut accéder à l'ensemble des membres publics de la classe InfoEtudiant (son interface statique) mais aussi à tous les membres publics de la classe Object qui n'ont pas été redéfinis par InfoEtudiant.

En utilisant l'instance infoEtudiant2, on peut accéder à l'ensemble des membres publics de la classe Object (son interface) et aussi aux méthodes redéfinies par InfoEtudiant via les mécanismes du polymorphisme.

L'ajout de nouvelles fonctionnalités ou l'interfaçage de celles déjà présentes peut être réalisé à l'aide de patrons de conception (par exemple décorateur ou adaptateur), mais cette application restera statique en tout point.

Nous appellerons une vue, une classe (ou un fragment de code) qui agira de cette façon, mais qui offrira une dimension importante, celle du dynamisme. La vue est en quelque sorte un filtre sur l'objet. Conceptuellement, l'objet est constitué d'un objet de base et d'un ensemble variable de vues. Conjointement, le regroupement de leurs méthodes publiques constitue l'essentiel de l'interface de communication avec l'objet d'application. On peut retrouver un premier exemple de vue à la figure 2.2.

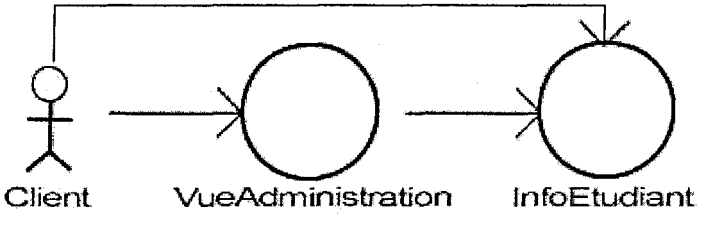

Figure 2.2 : Un premier exemple de vue

Dans l'exemple de la figure 2.3, la vue VueAdministration propose donc au client une interface d'utilisation fonctionnelle différente et indépendante de celle offerte par la classe InfoEtudiant et ce, sans aucune recompilation de l'objet de base. Un autre objet, sans aucun rapport avec le premier, pourrait aussi être utilisé avec une vue VueAdministration. Chacune des vues dérive d'un patron générique et abstrait appelé un point de vue. Les vues sont donc l'implémentation de points de vue pour une classe de base donnée.

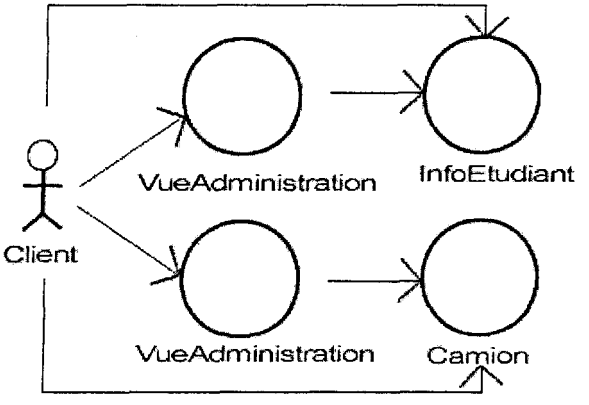

Figure 2.3 : Exemple simplifié du point de vue Administration

2.4.2 Objet de base et objet d'application

L'objet central sur lequel les vues sont appliquées se nomme objet de base (ObjBase). Dans notre cas, ça correspond à l'objet InfoEtudiant. Un objet de base peut être consulté directement ou à travers une ou plusieurs vues distinctes (figure 2.4).

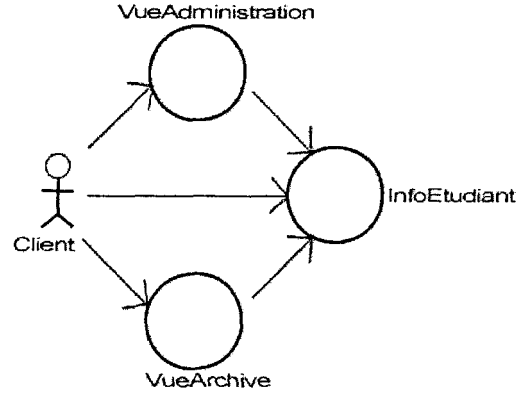

Figure 2.4 : Un objet de base peut être consulté de différentes façons (directement ou par ses vues)

Néanmoins, cette nouvelle couche d'abstraction ne s'ajoute pas sans inconvénients. Comment la vue VueAdministration devient-elle en relation avec l'objet InfoEtudiant? Comment peut-on gérer les accès simultanés à l'objet de base InfoEtudiant si on ne contrôle pas les vues? Contrairement aux interfaces traditionnelles, gérées par les mécanismes intégrés au langage, les vues sont des entités indépendantes de l'objet de base. Dans cette implémentation, les vues possèderont une référence sur l'objet de base, mais pas l'inverse. La solution consiste donc à insérer un objet plus global faisant office de passerelle entre le client et l'objet de base. On nomme cet objet l'objet d'application (ObjApp). Ses rôles principaux sont les suivants :

- Offrir un objet unique au client
- Dissimuler la gestion et la manipulation des vues
- Gérer les retours provenant de l'objet d'application
- Gérer la synchronisation des appels

2.4.3 Gestion des vues

Une fois l'instance de l'objet d'application en possession du client, ce dernier a la possibilité d'effectuer quelques actions particulières à la programmation par vue. Soit le code suivant :

Utilisation de l'objet d'application InfoEtudiantApp

```
InfoEtudiantApp infoEtudiantApp = ...;       // Création/instanciation de InfoEtudiantApp
infoEtudiantApp.attacherVue ("Administration");
infoEtudiantApp.attacherVue ("Archive");
boolean ok = infoEtudiantApp.estOK();
infoEtudiantApp.detacherVue("Administration");
```

Le client peut attacher les vues nécessaires selon ses besoins du moment, de façon à modifier les comportements de l'objet ou encore à en ajouter de nouveau. Une fois une vue attachée, on peut l'utiliser à travers l'objet d'application. Si plusieurs clients utilisent le même objet d'application, ils en partagent les vues actives. Cette situation peut devenir problématique lorsque la vue possède un état persistant associé à un client précis, par exemple des attributs. Supposons par exemple des clients C1 et C2 partageant la même vue VueAdministration à travers un même objet d'application et qui utilise au même moment la même méthode qui modifie la même ressource de l'objet de base. Ce partage de ressources peut occasionner des erreurs de synchronisation prévisibles, mais très laborieuses à gérer. Cet épineux aspect sera traité aux chapitres 4 et 5 avec l'intégration de la couche de sécurité.

2.4.4 La délégation des appels

Nous désignerons délégation, le mécanisme qui permettra de transmettre l'appel de méthodes d'une vue à l'objet de base et récriproquement. Habituellement, ObjApp

transfère l'appel à toutes les vues actives ayant défini la méthode demandée [Harrison et Ossher, 1993]. Dans certains cas, la délégation peut poser un ou plusieurs problèmes. Soit le cas suivant (représenté par la figure 2.5) :

- La vue VueFinance est active et contient la méthode paiement(...)
- L'objet de base (InfoEtudiant) et la vue ont chacun une méthode getBalance(...)
- Le client appelle la méthode paiement(...) de l'objet d'application (InfoEtudiantApp) qui redirige l'appel vers la vue active, soit VueFinance.

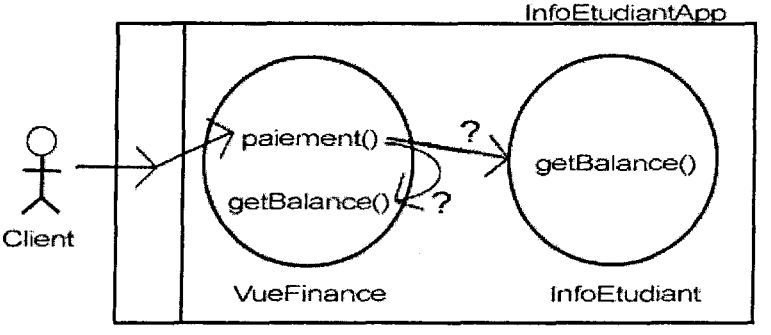

Figure 2.5 : Exemple de problème de délégation

Que faire si la méthode paiement(...) de la vue VueFinance fait l'appel à la méthode getBalance(...)?

Solution #1 : Délégation brisée *(*broken delegation - sécuritaire)

Dans ce cas, l'objet d'application se fie à la gestion de la vue pour effectuer l'appel de getBalance(...). L'exécution sera réalisée dans le contexte de la vue, on ne se rend pas à l'objet de base. Cela signifie que la méthode InfoEtudiant.getBalance(...) ne sera pas appelée par la vue VueFinance. Cette approche demande un peu plus de gestion mais,

respectant les règles internes de la vue, risque d'offrir une exécution plus sécuritaire (voir figure 2.6).

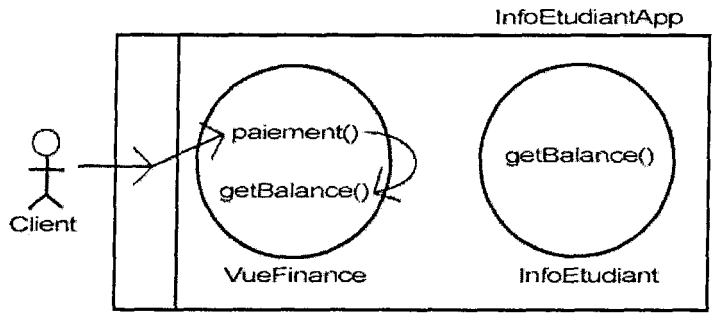

Figure 2.6 : Solution avec délégation brisée

La méthode paiement(...) avec délégation brisée

```
public void paiement (float montant){
        ...
        getBalance();
        ...
}
```

Solution #2 : Délégation partielle

Dans ce cas, l'objet d'application n'utilise pas l'interface de la vue pour effectuer son traitement (le pointeur *this*), il appelle directement getBalance(...) de l'objet de base. Cette solution est simple, mais non sécuritaire, car elle ne tient pas en compte la gestion fournie par l'objet d'application et par la vue. On donne plein pouvoir sur InfoEtudiant à chaque vue. Si la vue « VueFinance » possède getBalance(...), la méthode ne sera tout simplement pas appelée (voir figure 2.8).

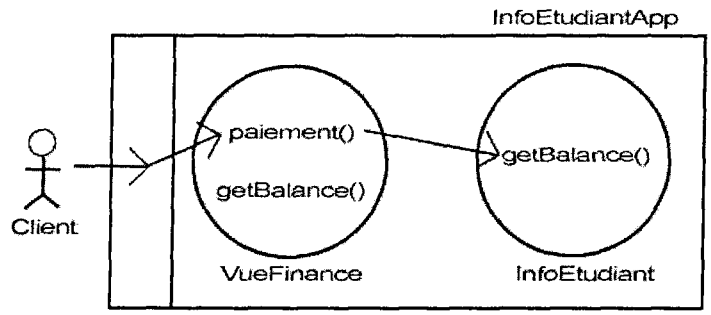

Figure 2.8 : Solution avec délégation partielle

La méthode paiement(...) sans délégation

```
public void paiement (float montant){
        objBase.setBalance(objBase.getBalance() - montant);
}
```

Solution #3 : Délégation respectée (non sécuritaire)

Dans ce cas, la vue retourne l'appel à l'objet d'application. Il s'agit de l'objet qui contient les règles de fonctionnement de l'object conceptuel.

Le problème du choix de l'implémentation reste entier, chaque méthode ayant des avantages et ses inconvénients. La délégation fera partie de la problématique reliée à l'intégration du modèle de sécurité où nous retrouverons une approche hyride de ces propositions.

2.4.5 La composition des vues

Avec la programmation par vue, il est possible qu'un client utilise l'objet de base à travers plusieurs vues très diversifiées les unes des autres. Les vues sont toutes en relation avec le même objet de base. Si deux vues sont attachées, le client peut appeler une méthode commune, par exemple getBalance(...). Mais laquelle des méthodes

getBalance(...) sera effectivement appelée? Celle de VueFinance, celle de VueArchive ou encore celle de l'objet de base? Et dans quel ordre?

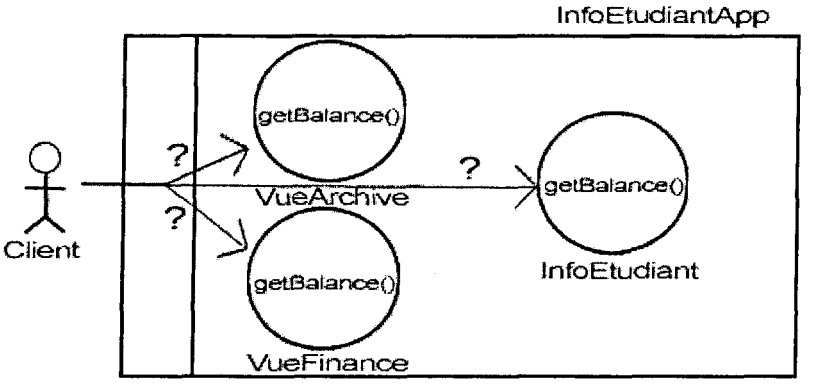

Figure 2.9 : Exemple de composition de vues

On appelle composition des vues, cette fonctionnalité de l'objet d'application qui permet l'encapsulation de cette implémentation. Dans InfoEtudiantApp, on pourrait voir un code ressemblant à ceci :

La méthode getBalance(...) de InfoEtudiantApp

```
public float getBalance() throws VOPException{
        float balance = getObjBase().getBalance();
        if (vueEstAttachee("Finance"))
                if (balance != getVueFinance().getBalance())
                        throw new VOPException("message au client");
        if (vueEstAttachee("Archive"))
                if (balance != getVueArchive().getBalance())
                        throw new VOPException("message au client");
        return balance;
}
```

L'implémentation de la méthode getBalance(...) dans la classe InfoEtudiantApp est prédéterminée et cette méthode peut être générée par les outils et techniques de programmation par aspect.

L'introduction d'un modèle de sécurité complexifie la gestion de la composition. On passe d'un modèle à usage général vers une gestion par usager qui rend cette approche inadéquate. Le chapitre 4 expose le problème et le chapitre 5 propose une solution.

2.5 Avantages de la programmation par vue

La programmation par vue apporte une multitude d'avantages qui ne sont pas associés à un langage de programmation particulier, mais bien à une méthode d'utilisation des objets. À la lumière de l'exploration des principaux concepts, présentons-en les trois plus marquants :

- L'avantage principal de la programmation par vue réside dans le fait que des fonctionnalités peuvent être ajoutées et retirées à un objet durant l'exécution, et ce, sans modifier l'environnement d'exécution. [Mcheick, 06]
- Développement décentralisé d'application. L'objet de base et ses vues enveloppent un objet de base indépendant.
- La réutilisation des points de vue permet l'homogénéité entre des systèmes ayant de vocations différentes. Par exemple, un point de vue Finance peut être appliqué à un étudiant dans système de facturation de frais de scolarité aussi bien qu'à un camion dans l'établissement de l'amortissement dans un budget.

2.6 Exemple

La programmation par vue est riche en terme de possibilités découlant des innombrables besoins fonctionnels. Il serait possible d'énumérer plusieurs cas d'intérêts

sans difficulté. Toutefois, pour limiter la portée de notre étude, un cas est présenté à la figure 2.10 : le profil étudiant avec la classe InfoEtudiant.

On peut y retrouver les divers aspects abordés en cours de chapitre. En bas à droite, l'objet d'application, contenant l'ensemble des vues et en relation avec InfoEtudiant, l'objet de base, situé à sa droite. La section supérieure illustre deux notions : l'implémentation des points de vue au dessus et la définition particulière des vues pour InfoEtudiantApp en dessous.

Pour des détails techniques supplémentaires, un exemple complet d'implémentation en Java est disponible à l'annexe A

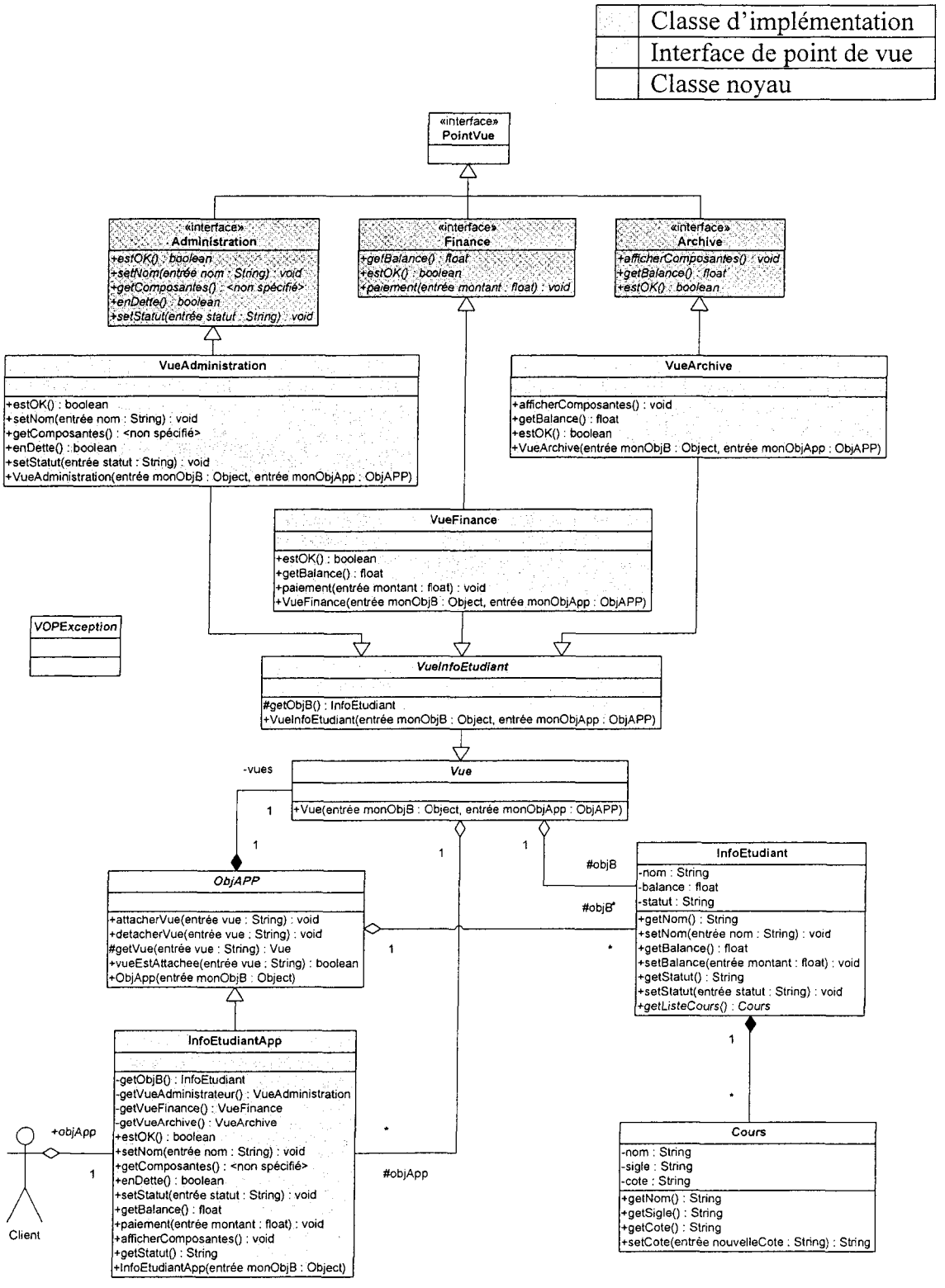

Figure 2.10 : Diagramme de classes de l'objet de base "InfoEtudiant"

Ce code à été réalisé avec les contraintes suivantes :

- Les classes doivent représenter avec précision les concepts théoriques de la programmation par vue
- L'objet de base ne doit pas être modifié par l'implémentation
- Les concepts des vues et de l'objet de base doivent être clairement dissociés de leur implémentation
- Développer un exemple concret, complet et fonctionnel afin de pouvoir y intégrer un modèle de sécurité

Avant de poursuivre avec la présentation de la programmation par vues, nous proposons, au chapitre suivant, de passer en revue le modèle de sécurité de Java qui sera utilisé dans notre solution.

3

LE MODÈLE DE SÉCURITÉ DE JAVA

La programmation par vue permet de gérer les variations fonctionnelles sur un objet de base contenu dans un système. L'idée que le comportement d'un objet puisse être modifié, en temps réel, par les besoins d'un client apporte maints questionnements. Par exemple, on se demande qui aura la responsabilité de modifier le comportement de l'objet d'application? On pourrait aussi vouloir connaître la source d'une demande afin d'en varier ou d'en limiter les actions sur l'objet de base. Chaque question où le client n'est plus anonyme, chaque question concernant le partage des ressources de l'objet d'application, chaque précision concernant la communication entre les objets entrainent sans équivoque l'intégration d'un modèle de gestion des authentifications et des autorisations. C'est ce que nous présentons dans ce présent chapitre concernant le modèle de sécurité de Java.

3.1 Java

Avant de plonger dans l'étude de notre problématique, survolons l'aspect sécurité du langage de programmation Java qui sera utilisé pour effectuer l'implémentation. Le choix d'un langage exclusivement objet s'est avéré plutôt naturel sachant que la programmation orientée objet (POO) est un moyen de réflexion permettant la résolution de problème et une

méthode d'organisation et de développement logiciel. De plus, l'approche de la programmation par vue est basée aussi sur l'objet.

En bref, Java a fait son apparition au milieu des années 90, parallèlement à l'explosion du Web et a été inventé par James Gosling de Sun Microsystems. Java est une évolution du C++ et est à la source du C#. Ses principaux avantages sont sa portabilité d'un système d'exploitation à l'autre, sa gratuité grâce à sa licence CDDL [SUN, 04], sa robustesse reposant sur une API solide, sans oublier sa convivialité de développement comparativement au C++.

Dans le cas présent, deux autres caractéristiques de Java sont importantes. La première est la simplicité avec laquelle il est aisé de créer des interfaces avec les mécanismes du langage. Ayant à l'esprit le défi de l'implémentation des vues dans l'objet d'application, les interfaces de Java, toujours abstraites et ne contenant aucune implémentation, représenteront adéquatement les promesses proposées par les points de vue. Elles seront utiles pour favoriser le comportement dynamique d'un objet.

La seconde caractéristique est critique et déterminante pour le projet. L'API de Java est étendue et elle contient, en plus de toutes les classes standards du langage, une série de classes dédiées à la gestion de la sécurité. Ce gestionnaire de sécurité sera à la source du modèle de sécurité de la programmation par vue.

3.2 Gestion des accès aux ressources

En plus des mécanismes intégrés de gestion des accès (public, private, protected, package), Java propose trois différentes couches de sécurité : compilation, vérification et chargement de classes.

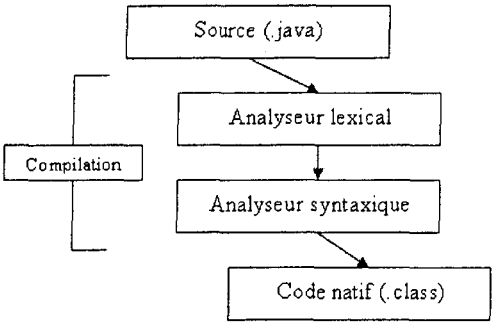

Figure 3.1 : Première couche de sécurité de Java (compilation)

La première couche de sécurité se retrouve à la compilation (voir figure 3.1). Les fichiers sources des classes (fichiers .java) sont évalués lexicalement et syntaxiquement par le compilateur. À ce niveau, Java produit du code natif appelé bytecode (les fichiers .class). C'est le code qui sera interprété par la machine virtuelle [Sebesta, 05].

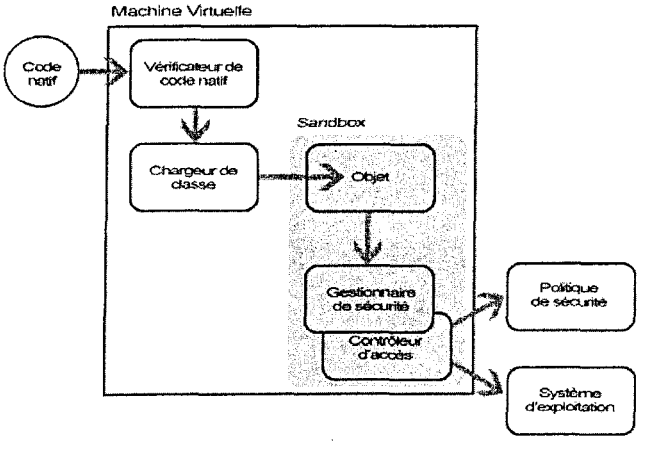

Figure 3.2 : Deuxième et troisièmes couches de sécurité de Java (vérification et chargement des classes)

La seconde vérification, telle qu'illustrée à la figure 3.2, est effectuée sur le code natif par un outil appelé le vérificateur de code natif *(bytecode verifier)*. Son rôle est de vérifier si le code de la classe répond aux spécifications de la machine virtuelle (*Java Virtual Machine – JVM*), aux règles d'héritage et de polymorphisme, aux conversions de type et de vérifier la mémoire nécessaire (gestion des *stack overflow*) [SUN, 06-5].

La dernière couche de sécurité est située à l'exécution. La classe est chargée en mémoire par le chargeur de classe (*class loader*) dans un contexte de sécurité tel que le sandbox qui est délimité par le gestionnaire de sécurité (*security manager*) [Oaks, 01].

3.2.1 Fonctionnement du sandbox

Le début de la sécurité de Java remonte à l'apparition des applets, petits programmes résidents qui sont téléchargés et ensuite chargés à l'intérieur d'une page Web par un navigateur. Afin de contrôler les accès de ces programmes fonctionnant sur le poste d'un client, un concept de bac à sable (sandbox) à été introduit (voir figure 3.3).

Ce sandbox représente en effet l'espace limité et contrôlé de ressources que peut utiliser un code Java non fiable [Plouin et al., 04]. Par exemple, il ne devait pas être possible de consulter un site, télécharger l'applet sur son poste et que cette dernière ait accès au disque dur ou aux fonctionnalités du système d'exploitation. L'impact aurait été catastrophique. Toutefois, les classes locales, c'est-à-dire hébergées sur la même machine, étant supposément fiables (trusted), pouvaient par conséquent accéder à l'ensemble des ressources locales.

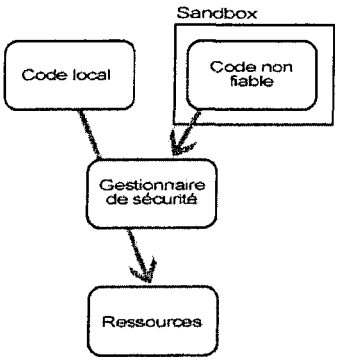

Figure 3.3 : Intégration du sandbox en Java

Un peu plus tard, afin de rendre le processus d'autorisation du sandbox plus flexible, un concept de signature à été introduit (voir la figure 3.4). Il consiste à signer un fichier d'archive de Java (fichier .jar) à l'aide d'une clé privée [SUN, 07].

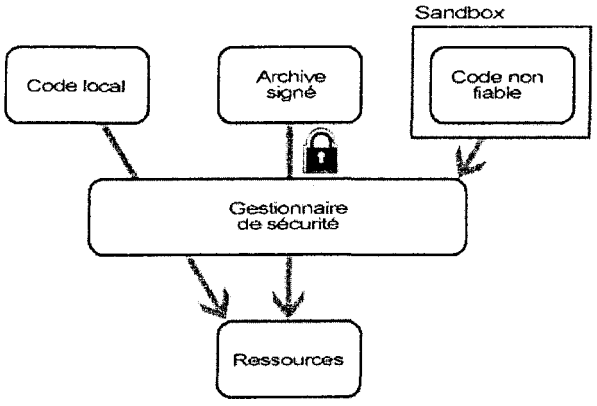

Figure 3.4 : Intégration des archives signées au gestionnaire de sécurité

Aujourd'hui, toujours par souci de flexibilité, l'évolution du sandbox a ajouté une nouvelle facette très utile : les politiques de sécurité (voir section 3.5.2). Une politique de sécurité permet à tout programme d'avoir des permissions particulières reliées aux fichiers, au réseau, au système d'exploitation, à la gestion des threads et à la sécurité. Les politiques

sont configurées localement sur la machine et peuvent varier. Dans ce cas, la sécurité est assurée par le gestionnaire de sécurité qui accède aux ressources du système d'exploitation à l'aide du contrôleur d'accès. Si on regroupe l'ensemble des permissions reliées à un code, on a donc un domaine de permissions qui peut être vu comme un sandbox personnalisé.

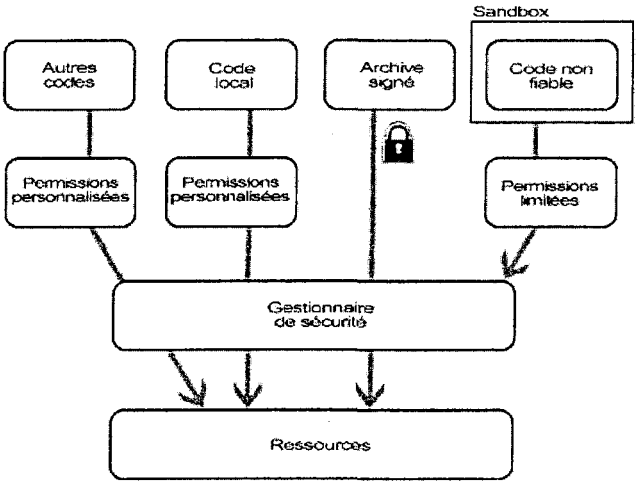

Figure 3.5 : Gestionnaire de sécurité complet

3.2.2 Le gestionnaire de sécurité

Le gestionnaire de sécurité (*security manager*) de Java a pour rôle de déterminer si l'exécution d'une opération particulière devrait être permise ou refusée. Par exemple, les applets Java sont sous le contrôle d'un gestionnaire de sécurité très strict qui définit les limites du sandbox. Il semble donc possible de bloquer les accès aux ressources du système aux applications Java. Cela n'est pas vrai dans tous les cas, car le gestionnaire de sécurité n'est pas démarré par défaut pour toutes les applications Java. Dans le cas des

applets, il est obligatoire. Autrement, le gestionnaire de sécurité doit être spécifié à la ligne de commande de démarrage de l'application via la ligne de commande suivante:

```
(Sécurité par défaut)
-Djava.security.manager

(En spécifiant le fichier contenant les politiques spécifiques)
-Djava.security.manager -Djava.security.policy==<FICHIER POLICY>
```

ou encore directement dans le code par l'opération

```
SecurityManager security = System.getSecurityManager();
```

Afin de mieux s'y retrouver, l'essentiel de l'algorithme que Java utilise pour valider la sécurité est illustré dans la figure 3.6.

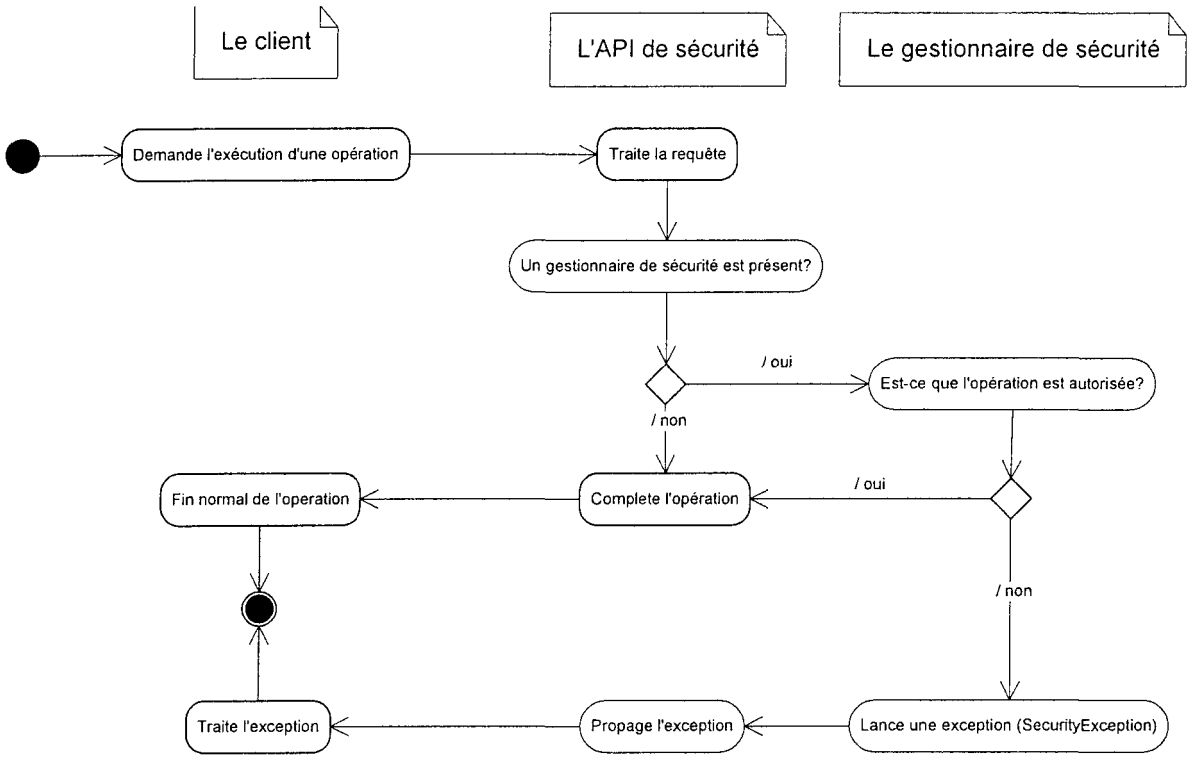

Figure 3.6 : Processus de sécurité de Java

3.2.3 Clés et signature

Afin de pouvoir identifier la classe appelante, la gestionnaire de sécurité a besoin d'information précise et sans équivoque sur l'appelant. Il peut le reconnaître au nom de la classe, à son emplacement sur le disque ou à son URL. Il peut aussi utiliser une méthode plus fiable reposant sur le concept de signature numérique. En Java, il est possible d'associer une classe ou une archive à une clé privée, par définition unique. À l'aide de la clé publique qui lui est associée, on peut s'assurer de la validité de la signature. Dans le cas qui nous intéresse, les objets pourront être signés par Java et le gestionnaire de sécurité leur assignera des permissions individuelles.

3.3 JAAS (*Java Authentication and Authorization Service*)

L'implémentation de la sécurité dans une application n'est jamais une affaire à prendre à la légère. La moindre faille peut entraîner la chute d'une application. Heureusement, pour les applications Java il existe JAAS. JAAS est intégré à l'API de Java depuis la version 1.4 [SUN, 06-1] et il a pour principaux rôles l'authentification et l'autorisation des usagers. Il intègre et rend plus transparente l'utilisation des mécanismes reliés au sandbox et au gestionnaire de sécurité.

En clair, si un appelant veut accéder à une application protégée par JAAS, il devra s'être préalablement identifié. Pas d'identité, pas d'accès! Une fois authentifié, le client est enfermé dans son sandbox où ses droits et privilèges sont restreints et contrôlés. Il est à noter que la plateforme JAAS n'est pas activée par défaut et qu'une programmation est nécessaire pour son fonctionnement.

Une application utilisant les classes d'authentification de JAAS aura un fonctionnement semblable à celui-ci :

1. Le programme veut s'authentifier
   - JAAS demande au programme de s'authentifier
     - Le programme fournit ses informations d'authentification
   - JAAS crée un objet d'identification (Subject) qui représente le programme
   - JAAS retourne l'objet d'identification au programme
2. Le programme veut exécuter une opération sécurisée
   - Le programme utilise la méthode statique doAs(...) de la classe Subject. Il envoie comme paramètres son objet d'identification et le code à exécuter.
   - La classe Subject utilise l'objet d'identification pour exécuter le code à la place du programme avec les permissions définies par JAAS lors de l'authentification.
3. Le gestionnaire de sécurité autorise l'opération
   - Le contrôleur d'accès vérifie si l'opération est valide vis-à-vis le domaine de permissions associé à l'objet d'identification
   - Si l'objet d'identification est autorisé, l'opération demandée par le programme est exécutée. Dans le cas contraire, une exception de type SecurityException est lancée.

La beauté du processus repose sur sa souplesse et sa simplicité. JAAS nous permettra de développer une relation <client, objet> plutôt que la traditionnelle relation <client,

<serveur, objet>> plus lourde et plus complexe. Reprenons donc plus en détail chaque étape pour mieux comprendre ce qui se passe réellement.

3.4 Processus d'authentification

Avant de définir des permissions et d'attribuer des privilèges, le client doit être reconnu. Il est nécessaire qu'il puisse être authentifié sans ambigüité. Avec la plateforme JAAS, l'authentification des clients est effectuée en sécurité et avec fiabilité indépendamment de qui est responsable de l'exécution du code [SUN, 06-1].

Un module JAAS possède la particularité d'être adaptable et indépendant de toute implémentation de technologie d'authentification. En effet, grâce à son architecture générique, l'implémentation peut être résolue seulement à l'exécution du programme (grâce à un fichier de configuration) ce qui en fait un excellent outil pour notre modèle de sécurité.

Le processus d'authentification se réalise à travers une séquence d'opérations appliquées sur une gamme d'objets spécifiques représentant l'entité à authentifier, ses identités particulières ainsi que le contexte et les outils de travail.

3.4.1 Entité et identité

Une entité est une personne ou un système qui interagit avec une ou plusieurs applications Java et qui peut prendre, pour chacune de ces interactions, une identité différente [Plouin et al., 04]. Sous JAAS, l'entité est représentée par la classe javax.security.auth.Subject et les identités (voir plus loin) par l'interface javax.security.Principal.

La classe Subject contient donc une liste d'objets Principal. Prenons l'exemple d'un citoyen. Le citoyen pourrait symboliser l'entité. On l'identifierait par son nom, son numéro de permis de conduire, son NAS, ses empreintes digitales, etc. Notre citoyen aurait plusieurs identités et celle utilisée dépendra de la situation.

En fait, une entité peut contenir deux types d'identité qui sont soit des identités d'information (Principal) ou soit des identités de sécurité (Credential) qui permettent d'assurer la légitimité du sujet (par exemple une clé privée). Dans le cas qui nous intéresse, seulement les identités d'information seront utilisées et le terme identité sera utilisé en ce sens.

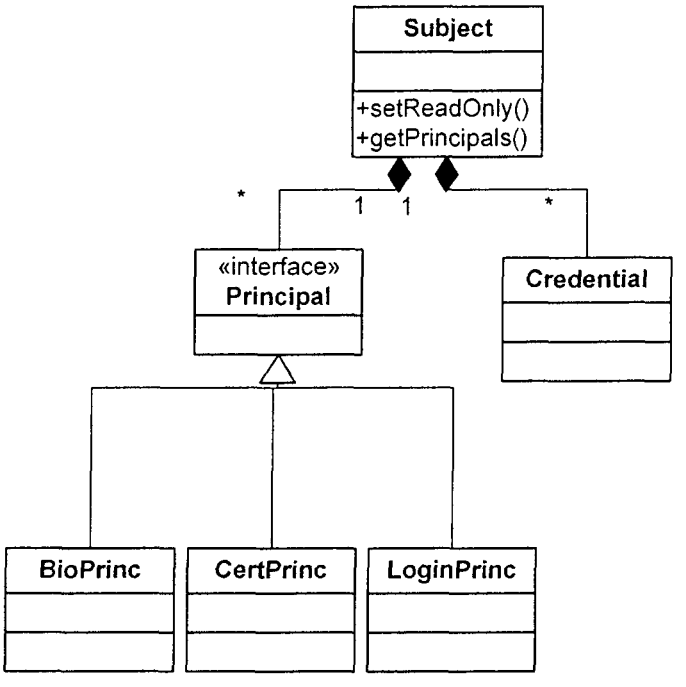

Figure 3.7 : Entité et identités

Le diagramme de classes de la figure 3.7 illustre les relations de composition et d'héritage entre les divers éléments que nous avons abordés jusqu'à présent. À la base, la classe Subject sera instanciée et complétée au cours du processus d'authentification un peu plus loin. L'interface Principal peut être implémentée de multiples façons. Dans cet exemple, la classe BioPrinc contiendrait une identification fournie par un système d'identification biométrique, la classe CertPrinc un certificat signé avec une clé RSA et la classe LoginPrinc un nom d'usager acquis grâce à un processus d'authentification traditionnel. Peu importe la manière dont l'utilisateur procure son identité, le processus d'authentification doit rester le même pour simplifier et standardiser [Plouin et al., 04].

La classe Subject contient deux méthodes d'intérêt : setReadOnly(...) et getPrincipals(...). Ces méthodes ont pour fonction respective de rendre le sujet non modifiable (sans possibilité de retour) et de retourner la liste d'identités lors du questionnement de l'entité.

3.4.2 Contexte d'authentification

Le contexte d'authentification représenté par la classe javax.security.auth.login.LoginContext fournit un moyen de développer une application indépendante des technologies d'authentification sous-jacentes [SUN, 06-4]. Le contexte encadre la portée effective des opérations effectuées par un processus appelant. On doit se souvenir que tout client, ou entité, opère à l'intérieur d'un cadre défini à l'instant qu'un gestionnaire de sécurité est présent. Il n'y a que deux alternatives, soit le client est authentifié et possède les droits en conséquence ou bien il ne l'est pas et ne possède que les

droits relatifs à un usager anonyme, ce qui revient finalement à dire qu'il ne possède que très peu de droits.

Le contexte est composé de tous les objets reliés à l'authentification (voir figure 3.8) mais il n'est pas responsable du processus lui-même. Il délègue l'ensemble des tâches à ses modules d'authentification, il leur fournit l'entité à compléter et le gestionnaire nécessaire aux communications. La classe Client est l'élément déclencheur du processus, il veut accéder à un objet sécurisé et le contexte réagit à sa demande.

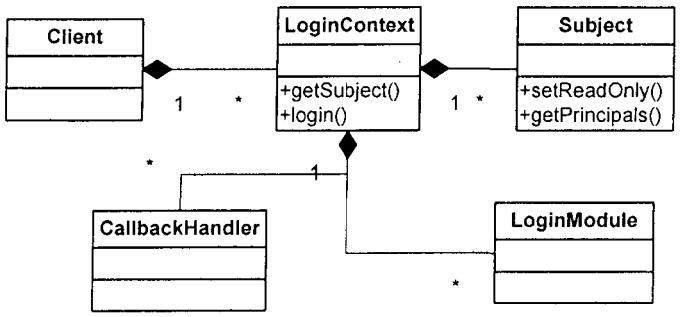

Figure 3.8 : Contexte d'authentification

L'utilisation du contexte d'authentification sera progressivement abordée au cours des prochaines sections.

3.4.3 Module d'authentification

Le module d'authentification de JAAS est composé d'une classe d'implémentation de l'interface javax.security.auth.spi.LoginModule (voir figure 3.9) et a pour rôle principal de fournir un modèle standardisé d'authentification.

Pour bien illustrer ce qu'est un module d'authentification, utilisons un exemple de la vie réelle. Lorsqu'un client d'une banque veut accéder au solde de son compte, il peut utiliser une des trois méthodes suivantes

1. Guichet automatique
2. Site web sécurisé
3. Directement au comptoir avec l'aide d'un préposé

Chaque méthode a ses particularités propres

- Authentification différente (ex : carte magnétique ou numéro d'identification)
- Possibilités d'entrée d'information (ex : clavier de guichet ou clavier d'ordinateur)
- Possibilités de sortie (ex : moins d'options disponibles au guichet que sur le site)

Dans cet exemple, on peut s'authentifier à l'aide de modules d'authentification distincts et la communication avec le client dépend du module utilisée (voir gestionnaire de communication à la section 3.4.4)

JAAS nous offre de produire des modules d'authentification personnalisés dont le code Java qui le compose reflète les particularités du système. À la différence de notre exemple précédent, il est possible de combiner une série de modules d'authentification.

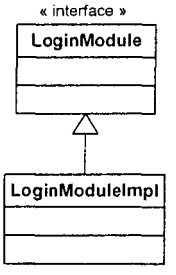

Figure 3.9 : Implémentation du « LoginModule »

Lorsqu'une demande d'authentification est engagée, tous les modules d'authentification associés sont chargés dans le contexte d'authentification courant. La liste des modules est contenue dans un fichier de configuration ce qui a pour conséquence de conserver une

distance face au système. L'entité (Subject) est créée à ce moment (ou encore reçue en paramètre) et ses identités seront ajoutées par les différents modules lors de l'exécution. Une série de paramètres d'option peut être associée aux modules lors de leur instanciation. Voici la liste des principales options :

- REQUIRED : L'exécution de l'authentification de ce module est obligatoire. En cas d'échec, le module suivant est tout de même appelé.
- REQUISITE : L'exécution est obligatoire, mais en cas d'échec, le processus d'authentification s'interrompt.
- SUFFICIENT : L'exécution avec succès de ce module est suffisante pour conclure le processus d'authentification.
- OPTIONAL : Ne conditionne ni l'authentification du client ni l'appel du module suivant.

Ces options seront utilisées plus loin pour compléter l'authentification.

De l'extérieur, le processus d'authentification est perçu comme un seul bloc d'opérations linéaires mais, en fait, il s'agit plutôt d'une opération en trois phases distinctes : l'initialisation (voir figure 3.10), l'authentification (voir figure 3.11) et la validation (voir figure 3.12).

Phase #1 : Initialisation

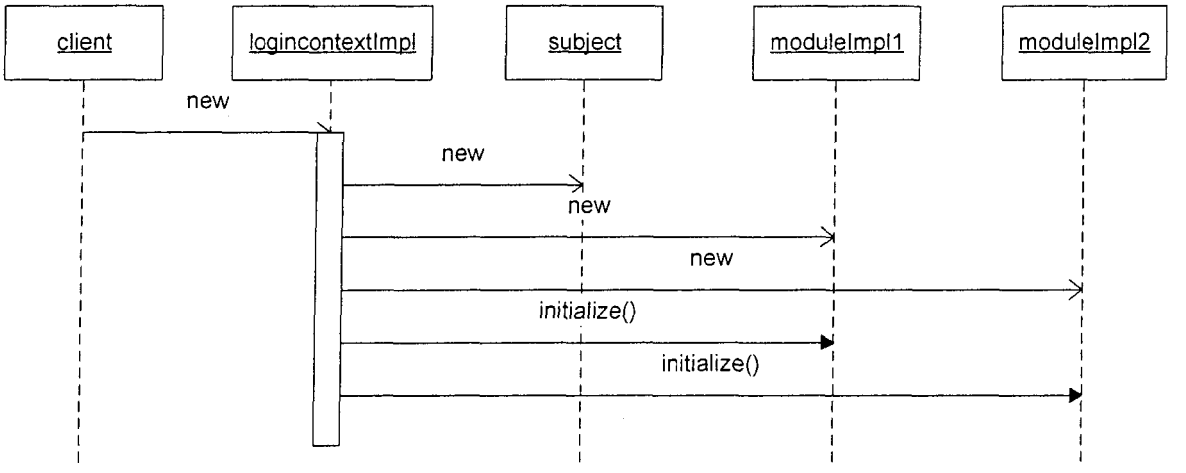

Figure 3.10 : Authentification, phase #1

Le processus d'initialisation est enclenché par le client qui instancie le contexte d'authentification. Le contexte instancie à son tour ses modules d'authentification et, s'il ne l'a pas déjà reçu en paramètre, une nouvelle entité vide. Il n'y a pas de limite quant à la quantité et à la diversité des modules. Chaque module redéfinit la méthode initialize(...) qui a pour rôle de le préparer à l'authentification. Cette méthode remplace en fait le constructeur. Il est à noter que le client est totalement isolé de ce processus interne.

On peut retrouver une explication plus complète sur le site de Sun [SUN, 06-2].

Phase #2 : Authentification

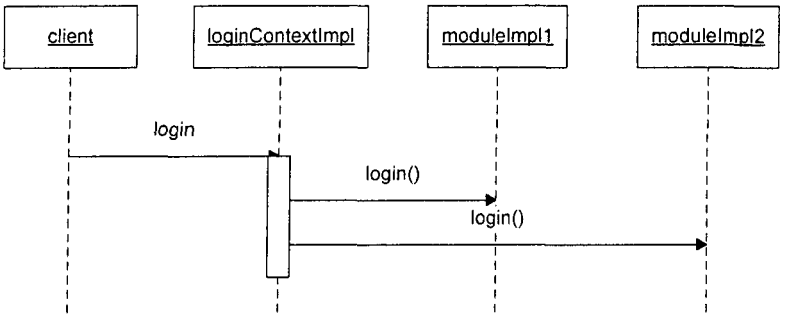

Figure 3.11 : Authentification, phase #2

Le client décide maintenant de débuter le processus d'authentification avec l'appel de la méthode login(...), sans paramètre. Le contexte fait de même avec une série d'appels à la méthode login(...) de chacun de ses modules. Les retours peuvent être positifs (un succès), négatifs (un échec) ou encore une exception (un problème). C'est à ce moment que les options d'initialisation des modules auront une influence déterminante sur quels modules seront appelés. À cette étape, les modules ne modifient pas l'entité, mais conservent plutôt le résultat de leur authentification dans leur état privé.

On peut retrouver une explication plus complète sur le site de Sun [SUN, 06-3].

Phase #3 : Validation

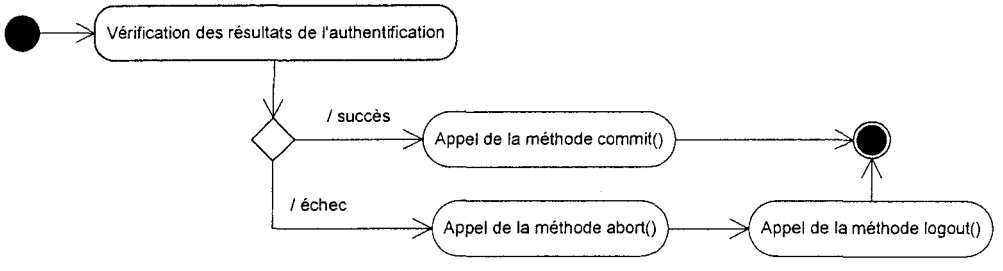

Figure 3.12 : Authentification, phase #3

Finalement, dépendamment des options définies et des résultats de l'authentification, le contexte appelle soit la méthode commit(...) ou encore la méthode abort(...) des modules. La première méthode modifie l'entité en conséquence, par l'ajout d'une identité. La seconde méthode analyse la raison de l'échec de la validation et peut, au besoin, faire l'appel de la méthode logout(...) ce qui a pour effet de démunir l'entité de toutes ses identités et de la rendre non modifiable pour le reste de l'exécution.

3.4.4 Gestionnaire de communication

Lorsqu'un module d'authentification a besoin d'information que les systèmes informatisés avec lesquel il est relié ne peuvent lui fournir, il lui est nécessaire de communiquer avec le client ou tout autre système externe. En Java, cette opération de retour vers l'appelant est représentée par l'interface javax.security.auth.callback.Callback. Chacune des implémentations de cette interface définit une voie de communication adaptée entre le système et le client. Un *callback* ne récupère et n'affiche pas les informations requises par les divers services de sécurité [SUN, 06-4]. Ils fournissent simplement les méthodes d'entrée et de sortie pour l'information demandée.

La classe NameCallback pour les noms d'usager, la classe PasswordCallback pour les mots de passe et la classe TextOutputCallback pour l'affichage ne sont que quelques exemples d'implémentation que l'on retrouve dans l'excellente API de Java. Il est aussi possible d'écrire ses propres classes de retour qui pourraient encapsuler toute forme de communication nécessaire.

Pour gérer les communications lors de l'authentification, les classes de retour sont contrôlées par un gestionnaire de communication nommé javax.security.auth.callback.CallbackHandler. Cette interface possède une seule méthode soit handle(...) qui reçoit les objets Callback et décide ce qu'il doit en faire (représentation à la figure 3.13). Des exemples de codes seront fournis à travers le chapitre 5 et se retrouvent aussi en annexe.

Son rôle au niveau de l'authentification est capital. Il assure que les modules d'authentification puissent communiquer entre eux et avec le client.

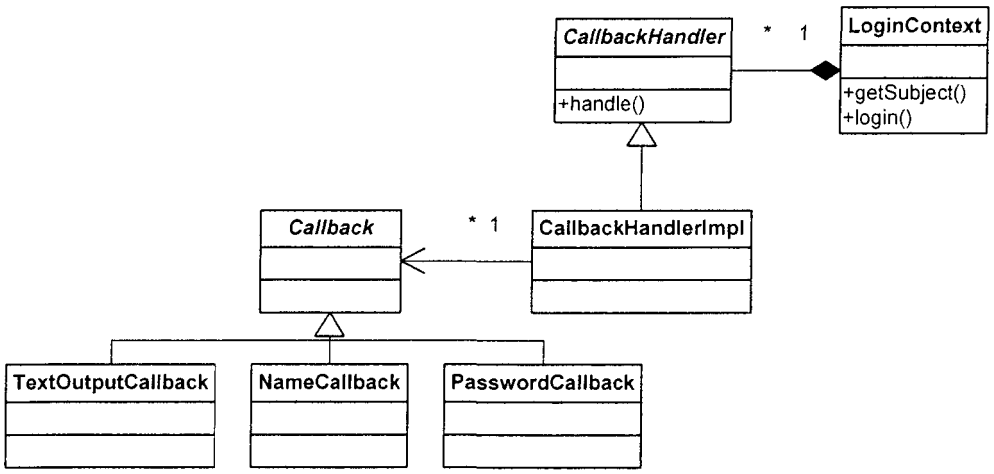

Figure 3.13 : Classes permettant la communication lors de l'authentification

3.5 Processus d'autorisation

Le processus d'autorisation s'effectue à la suite de l'authentification. À l'aide des identités incluses dans l'entité, il sera maintenant possible d'appliquer des restrictions à notre objet identifié. L'autorisation sous JAAS étend l'actuelle architecture de sécurité de Java qui fait usage de politiques de sécurité pour spécifier les droits et privilèges au code

qui s'exécute. JAAS propose une approche centrée sur le client (figure 3.15) et non sur la provenance du code (figure 3.14), il nous permet de modifier la source et les caractéristiques de la source de l'appel afin de permettre l'exécution d'opérations sécurisées [SUN, 06-3].

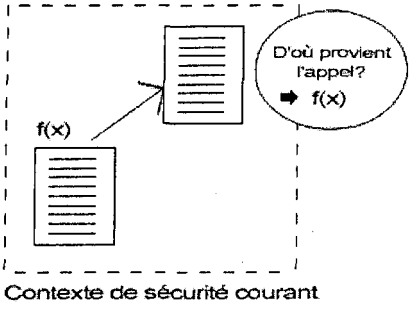

Figure 3.14 : Validation de sécurité selon la provenance du code

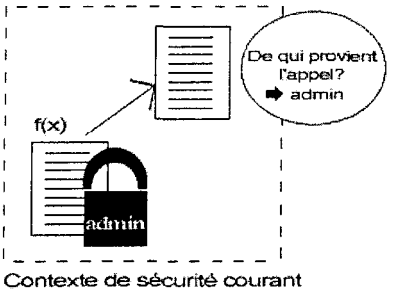

Figure 3.15 : Validation de sécurité selon le client

3.5.1 Les permissions

Une permission (java.security.Permission) est une classe abstraite qui représente l'accès à une ressource du système. Par exemple, la classe FilePermission permet de contrôler l'accès à un fichier ou un dossier et la classe SocketPermission définit les permissions sur

un liste des ports de communication sur lesquels on veut restreindre les accès. On pourrait même vouloir contrôler l'accès aux instances d'une classe particulière, par exemple une vue, et créer une nouvelle permission personnalisée en conséquence. Résultat : tout accès à l'objet devrait avoir été autorisé par le gestionnaire de sécurité! Nous reviendrons sur l'implémentation de cette permission au chapitre 5.

Une caractéristique importante des permissions est qu'elles doivent être octroyées à quelque chose, comme à une entité de type Principal par exemple. Supposons qu'un client s'authentifie et qu'une identité représentée par la classe ExemplePrincipal soit ajoutée à son entité (la classe Subject) sous le nom de *exemple*. Une permission pourrait être créée et elle pourrait se traduire par cette règle :

SI l'identité reliée à l'exécution courante dans le contexte courant possède une identité nommée *exemple* qui est de type « ExemplePrincipal »

ALORS accorder la permission

SINON Refuser l'accès

Les permissions à vérifier sont inscrites dans le fichier des politiques de sécurité.

3.5.2 Les politiques de sécurité

Les politiques de sécurité de Java sont consignées dans le fichier *<dossier d'installation Java>/jre/lib/security/java.policy*. Il s'agit des politiques de sécurité par défaut qui sont chargées par le gestionnaire de sécurité. On peut modifier ou remplacer ce fichier car il est, avec raison, très restrictif. À l'exécution, la JVM regroupe dans des domaines de protection l'ensemble des classes dont les instances partagent le même jeu de

permissions. Un domaine de protection peut être l'équivalent d'un sandbox [Plouin et al., 04].

Ce fichier de politiques de sécurité est constitué d'une ou de plusieurs sections d'attributions, qui commencent par le mot clé grant. Voici un exemple de la structure que peut avoir ce fichier de configuration :

Fichier de configuration java.policy

```
grant signedBy "signataire" codebase "URL",
principal principal_class_name "identité1",
principal principal_class_name "identité2",
{
        …
        permission permission_class_name "cible1", "action", signedBy "signataire";
        permission permission_class_name "cible2", "action", signedBy "signataire";
        …
};
```

Le mot-clé grant permet d'attribuer des permissions au code Java sur la base de trois critères optionnels :

- Le signataire du code (signedBy) : Si le code est signé par la ou les utilisateurs indiqués, les permissions spécifiées par le grant lui sont accordées.
- L'emplacement du code (codebase) : Si le code est chargé depuis l'URL spécifié, alors les permissions spécifiées lui sont attribuées.
- L'identité sous laquelle le code est exécuté (principal) : Si le code est exécuté avec une identité du type principal_class_name et de nom identité1 alors les permissions spécifiées lui sont attribuées.

Dans l'exemple qui suit, on peut comprendre que l'identité *exemple* de type ExemplePrincipal, une classe incluse dans l'archive Exemple.jar a la permission de lire

(l'action *read*) la propriété système *java.home* et *user.home* ainsi que de lire le fichier *secret.txt*. Tout autre accès et ce pour tout usager est bloqué.

Exemples de permissions

```
grant codebase "file:./Exemple.jar", principal ex.principal.ExemplePrincipal "exemple"{
        permission java.util.PropertyPermission "java.home", "read";
        permission java.util.PropertyPermission "user.home", "read";
        permission java.io.FilePermission "secret.txt", "read";
};
```

3.5.3 Fonctionnement

Après ce rapide survol de la syntaxe, il reste encore à réaliser l'action la plus importante : l'association entre l'entité et ses permissions au cours de l'exécution. Comment l'appel d'une méthode provenant d'une classe anonyme déclenche-t-il la validation puis l'autorisation d'accès à une ressource? La clé est la classe Subject.

Subject possède une méthode statique doAs(...) avec comme paramètres l'instance de l'entité (javax.security.auth.Subject) et une action privilégiée (java.security.PrivilegedAction). La méthode doAs(...) associe l'entité fourni avec le contexte de sécurité courant (anonyme) et appelle ensuite la méthode run(...) de l'action privilégiée. Cette méthode run(...) doit contenir tout le code qui doit être exécuté avec comme instigateur masqué l'entité reçue en paramètre. En fait, JAAS s'assure que l'action soit finalement exécutée par l'entité (Subject) à l'intérieur de son propre bac à sable ou, dans d'autres termes plus techniques, son contexte de sécurité [SUN, 06-3]. La figure suivante illustre la gestion des accès avec JAAS (un exemple de code sera abordé à la section 5.5.2).

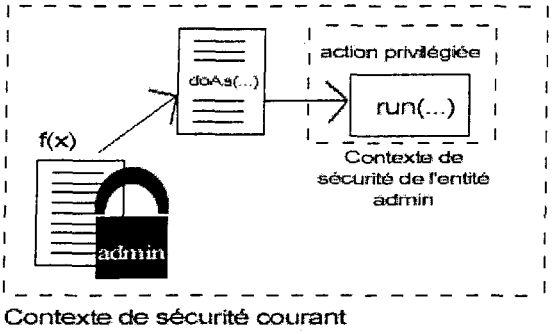

Figure 3.16 : Gestion des accès avec JAAS

Pour notre étude, nous nous limiterons à cette interaction mais le modèle de sécurité de Java est d'une plus grande envergure.

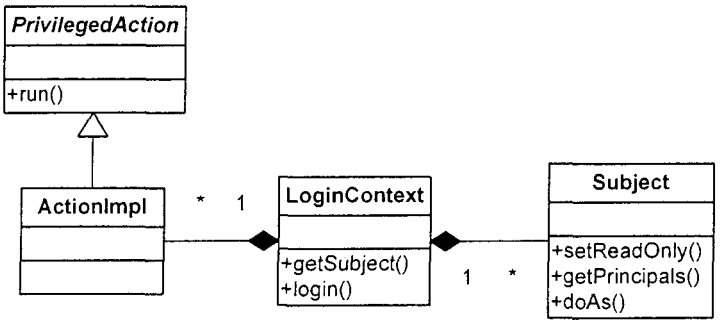

Figure 3.17 : Classes utilisées lors du changement de contexte de sécurité

Finalement, le fonctionnement du processus d'autorisation peut être résumé comme suit:

- Le client effectue une action sous une identité différente
- Le gestionnaire de sécurité se charge de valider les opérations effectuées sous cette identité
- Dès que l'action privilégiée tente d'accéder à une ressource protégée, le gestionnaire vérifie les politiques de sécurité

- Si on demeure dans le bac à sable, on peut continuer
- Si on sort du bac à sable, le gestionnaire de sécurité nous indique qu'on dépasse nos limites et lance une exception de violation de sécurité.

3.6 Modèle de sécurité en UML

La sécurité en Java est beaucoup plus complexe que ce qui vient d'être abordé, nous en avons survolé un minimum dans ce bref document. La partie reliée au code Java sera reprise et développée au chapitre 5. Sun Microsystems offre des tutoriaux très complets et très clairs sur lesquels le diagramme de classe a été modélisé [SUN, 06-1] [SUN, 06-2] [SUN, 06-3]. Afin de mieux s'y retrouver, un aperçu un peu plus global et technique des idées apportées en cours du chapitre est fourni à la figure 3.18. Avec ce diagramme de classes, il est possible d'y retrouver un exemple des relations qui existent entre les classes permettant l'authentification et l'autorisation. Noter que seulement les principaux éléments ont été représentés.

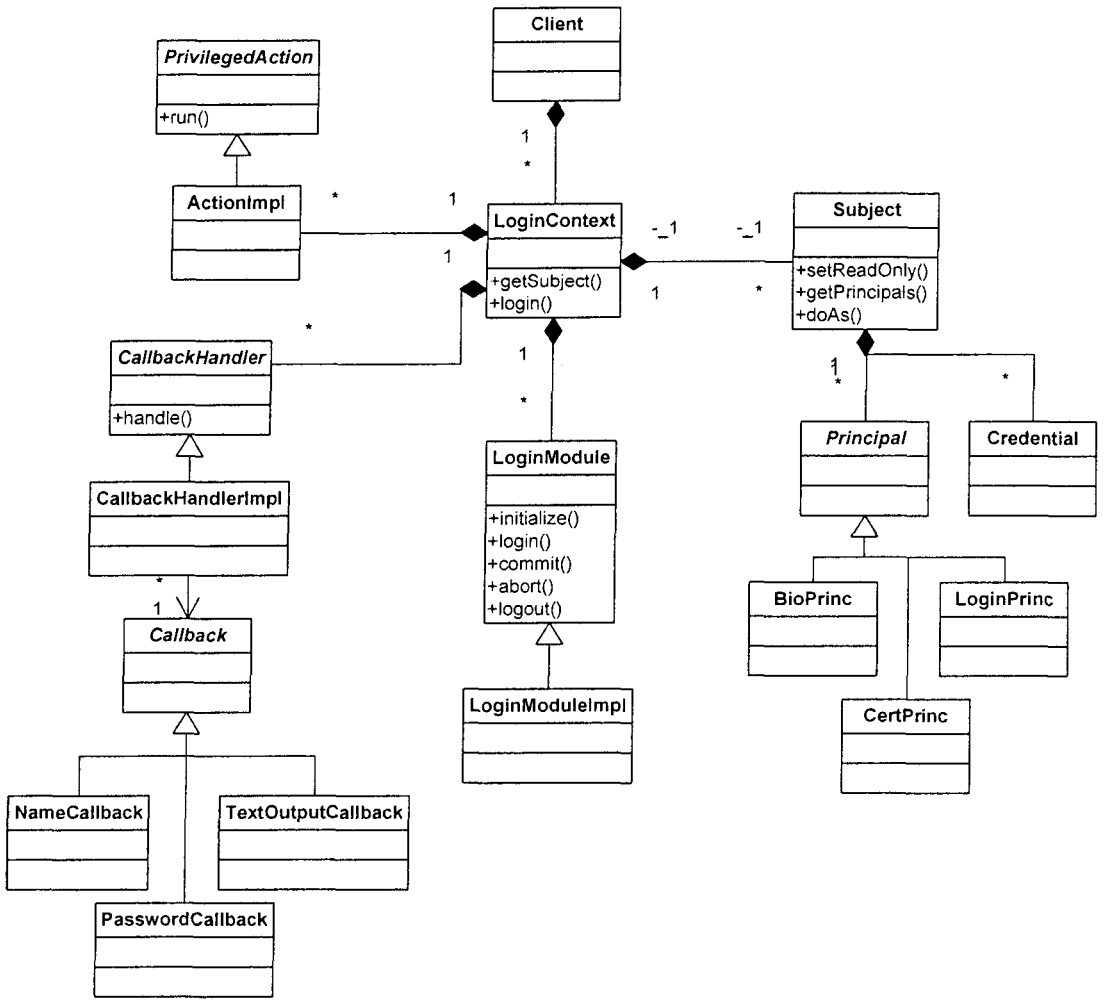

Figure 3.18 : Principales classes de l'authentification et de l'autorisation (Sun Microsystems)

Les éléments de JAAS que nous n'avons pas abordé n'étaient pas, à première vue, utiles à notre solution. Le chapitre suivant réutilisera le même modèle, on peut déjà y remarquer les éléments évolutifs du modèle représentés par les classes d'implémentation (par exemple, LoginModuleImpl et ActionImpl). Notre solution proposera de pratiquer la greffe de la programmation par vue sur ce modèle de sécurité.

4

ÉNONCÉ DE LA PROBLÉMATIQUE

4.1 Énoncé

Comme nous l'avons abordé au chapitre deux en introduisant les concepts de la programmation par vue, un système informatique doit évoluer à travers le temps pour répondre aux nouveaux besoins des clients qui l'utilisent. La programmation par vue propose une solution dynamique permettant d'encapsuler un système existant, possiblement victime d'une évolution pénible. On se rappelle qu'on ajoute une série d'objets d'application intermédiaires qui cachent des vues fonctionnelles évolutives communiquant avec un objet de base contenu dans le système désuet.

La programmation par vue a déjà été implémentée par deux prototypes dirigés par M. Mili, à l'aide du langage C++ [Dargham, 01] et aussi en Java [Mcheick, 06]. La problématique principale ne sera pas particulièrement reliée à un langage en particulier toutefois nous utiliserons le langage Java en raison de son architecture de sécurité bien adapté à notre besoin. Nous développerons autour de cette implémentation.

En fait, la problématique n'est pas l'approche de la programmation par vue, il s'agit plutôt de tenter de spécialiser une section du modèle. La problématique découle justement

de l'extension de la programmation par vue, de l'ensemble des impacts occasionnés par l'ajout d'un modèle de sécurité. Nous les regrouperons en cinq sous problématiques soit :

- La problématique de l'approche actuelle (voir section 4.2) : Elle regroupe tous les problèmes, énoncés dans les chapitres précédents, résultant de modifications réalisées sur notre propre implémentation de la programmation par vue.
- La problématique de délégation (voir section 4.3) : Discussion au sujet d'une problématique soulevée par Hamid Mcheick et plusieurs autres [Harrison et Ossher, 1993] [Mili et al., 01] [Mcheick, 06] se rapportant à la sécurité des appels.
- La problématique de sécurité (voir section 4.4) : Le cœur du sujet, l'identification des intervenants et la gestion des accès aux vues.
- Problématique de synchronisation (voir section 4.5) : Présentation d'une nouvelle problématique sur la gestion simultanée des accès.
- Problématique reliée aux exécutions partielles (voir section 4.6) : Présentation d'une nouvelle problématique concernant les difficultés de retour en arrière à l'intérieur d'un système contenant des ressources partagées.

Avec le modèle de sécurité de la programmation par vue en Java, nous tenterons de développer un prototype d'auto-adaptation des objets afin de libérer le client de la responsabilité, et du pouvoir, de déterminer lui-même ses accès aux vues et de mieux contrôler ses droits sur l'objet de base lui-même.

S'ajoutant à la problématique elle-même, nous nous sommes fixés une série de contraintes technologiques qui auront pour rôle de circonscrire le développement et de

s'assurer que l'extension de sécurité ne contrevienne pas aux objectifs de base de la programmation par vue. Les voici donc :

- Le programme doit rester simple et léger afin de préserver les droits d'accès et la facilité d'utilisation pour tout programme client
- L'ajout du module doit être le plus transparent possible pour l'usager
- L'extension doit être solide et efficace (sécurité de Java). Limiter l'utilisation de profil à sa plus simple expression
- Conserver les rôles des objets et des relations déjà existants dans la programmation par vue

L'ensemble des problématiques soulevées dans le présent chapitre sera revu et discuté au chapitre 6 à la lumière de la solution proposée au chapitre 5.

4.2 Approche actuelle de la programmation par vue

Approfondissons la description de la problématique à l'aide des éléments de la programmation par vue. Avant de transformer le système actuel, il serait judicieux d'analyser son fonctionnement et de se poser la question suivante : est-ce que l'ajout d'options de sécurité influence sur ses fonctionnalités?

Sans chercher longtemps, on peut rapidement apercevoir quelques nuages noirs à l'horizon...

4.2.1 Le rôle du client

Soit un objet d'application ObjApp composé des vues V1 et V2 qui est relié à un objet de base ObjBase. Jusqu'à présent, le client était unique et totalement en contrôle de l'objet

d'application. Le rôle du client était de s'assurer que les vues nécessaires à la réalisation d'un besoin soient attachées et actives. Prenons l'exemple de la figure 4.1, si un client C1 veut appeler la méthode f(x) de V1, il doit préalablement s'assurer que la vue V2 soit active car V1.f(x) fait appel à V2.g(x)

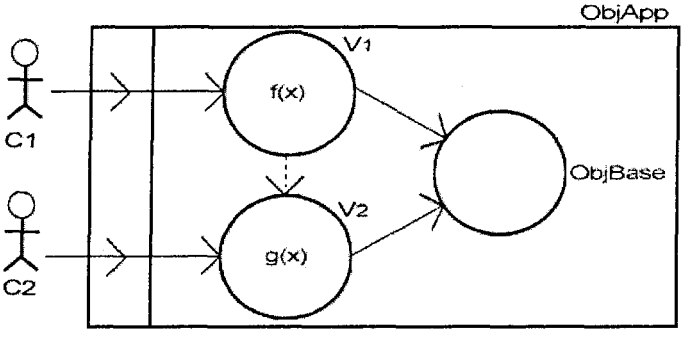

Figure 4.1 : Exemple d'un ObjApp avec deux vues

Que se passerait-t-il si un second client C2 utilise aussi ObjApp? Si ce dernier désire lui aussi exécuter V1.f(x) mais sans avoir activé V2, par choix ou par négligence, l'appel fonctionne correctement! Supposons que C2 décide par la suite de détacher la vue V2? Se pensant en contrôle, C1 appelle à nouveau C1.f(x) et est très confus par la réponse de ObjApp (Il reçoit une exception). Entre deux appels successifs de f(x), où l'objet d'application n'a pas été utilisé par le client, les comportements diffèrent. C2 aurait aussi des problèmes, la désactivation de la vue V2, qu'il n'a pas activée lui-même, empêcherait l'appel de V1.f(x). Mystère…

Supposons encore qu'il n'existe qu'un objet d'application partagé par tous ses clients. Le client devrait alors questionner ObjApp avant chaque appel, histoire de s'assurer qu'il

est utilisable. Il s'agit d'une solution simpliste et trop lourde qui va à l'encontre des limitations de simplicité de la programmation par vue.

Énonçons plutôt la problématique suivante « **Le rôle du client n'est pas adapté à une approche multi clients** ».

Notons toutefois que l'approche initiale de la programmation par vue supposait qu'une vue ne pouvait dépendre des fonctionnalités d'une seconde vue, elles sont indépendantes. Notre approche actuelle élargi la définition. Aussi, nous évitons de discuter d'une approche distribuée qui gère ce problème mais qui demande un serveur d'application pour fonctionner [Mcheick, 06].

4.2.2 Persistance des vues

Avec un seul client, la question de persistance des vues ne se posait pas. Les vues sont créées et détruites au besoin. Prenons l'exemple de la figure 4.2, une vue V1 partagée entre deux clients C1 et C2. Cette vue possède un attribut nommé valeur et des méthodes f(x) et g(x).

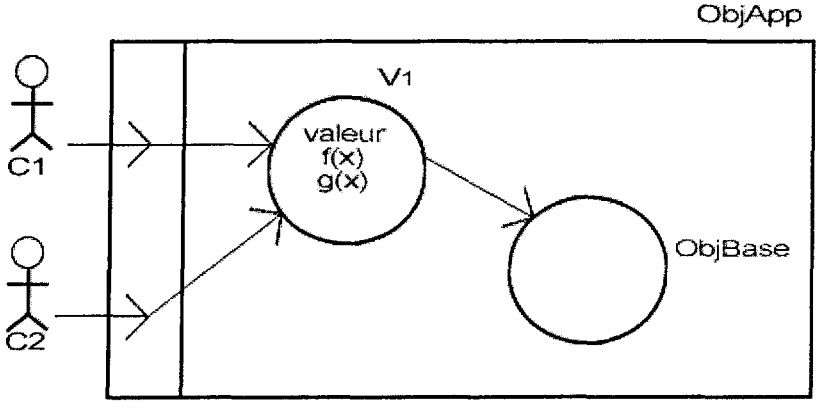

Figure 4.2 : Exemple d'un ObjApp avec une vue V1 partagée entre 2 clients

Supposons que le résultat de l'appel de g(x) modifie l'attribut valeur.

Supposons que f(x) puisse modifier ObjBase (cette modification dépend de la valeur de son attribut valeur).

Considérons la suite d'opérations suivantes :

À t0, C1 appelle V1.f(x) et influence ObjBase

À t1, C2 appelle V2.g(x) et modifie valeur

À t2, C1 appelle V1.f(x) et, surprise, rien ne se passe

Dans ce cas, le changement de l'attribut d'une vue, de son état en fait, crée des problèmes qui pourraient entrainer des changements de comportements inquiétants. Avec l'ajout d'une sécurité qui identifierait C1 et C2 dans ObjApp, comment pourrait-on gérer cette problématique? Pourrait-on envisager de transformer nos vues en objet s'apparentant à des EJB (StatefulBean et StatelessBean)? Ainsi la problématique « **La persistance de la vue est représentative des actions d'un seul client. Le partage des informations est source de confusion** » doit être considérée dans l'intégration de notre modèle de sécurité.

4.2.3 Permanence du profil

Les restrictions imposées à la section 4.1 mentionnent qu'on doit garder notre objet d'application léger et facile d'utilisation, ce qui exclut toute forme d'installation et de configuration du côté client. Avec la programmation par vue, nous avons une relation <client, ObjApp> et on veut éviter la transformation vers une relation <client, <serveur, ObjApp>>. Alors, avec l'ajout d'option sur la sécurité, où se situera l'inévitable profil contenant les informations du client?

Chez le client? Il devra alors fournir son profil à chaque communication!

Dans l'objet d'application? Il devra connaître les besoins et l'état de tous ses clients!

C'est une épineuse problématique qui sera en partie reprise à la section 4.4. Pour l'instant, soulignons seulement que « **Le modèle actuel n'assure pas la permanence du profil usager** ».

4.2.4 Transparence d'utilisation des vues

Pour être utilisée, une vue doit être attachée et ensuite activée. Elle pourra être désactivée, rendue inaccessible, et finalement détachée. Cette double condition limitant l'utilisation de ObjApp va à l'encontre des la contrainte de simplicité. Ajoutons seulement la problématique suivante : « **La dualité attachement/détachement et activation/désactivation du processus de gestion des vues représente une réalité informatique technique qui ne devrait pas être visible du côté du client** »

Avec l'ajout d'un modèle de sécurité, on pourrait peut-être altérer cette approche tout en conservant la philosophie de la programmation par vue.

4.2.5 Évolution dynamique

Nous savons déjà que les vues sont utilisées pour représenter des points de vue particuliers concernant un aspect fonctionnel spécifique d'un objet de base. Il se peut que pour bien représenter un besoin à un moment donné, il faudra utiliser plusieurs vues.

Soit ObjApp a t0 avec trois vues actives V1, V2 et V3 qui sont utilisées lors de la séquence d'appels initiée par ObjApp.f(x). On appelle donc V1.f(x), V2.f(x) et V3.f(x) ce qui provoque une modification particulière de ObjBase (voir figure 4.3).

Soit une évolution des besoins entre t0 et t1 qui nécessite de désactiver les vues V2 et V3 et de conserver V1. L'appel de ObjApp.f(x) n'utilise plus que V1.f(x) (voir figure 4.4).

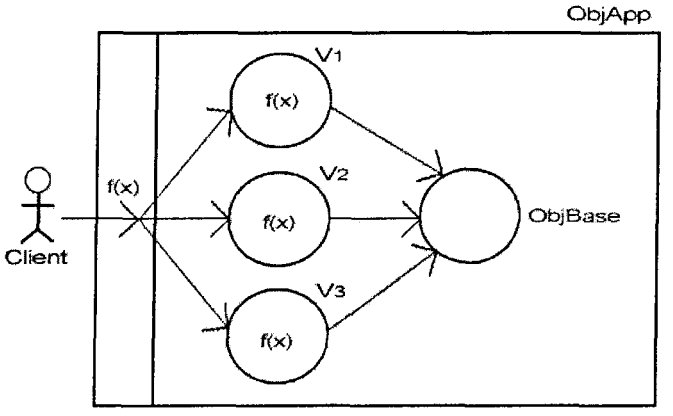

Figure 4.3 : ObjApp à t_0

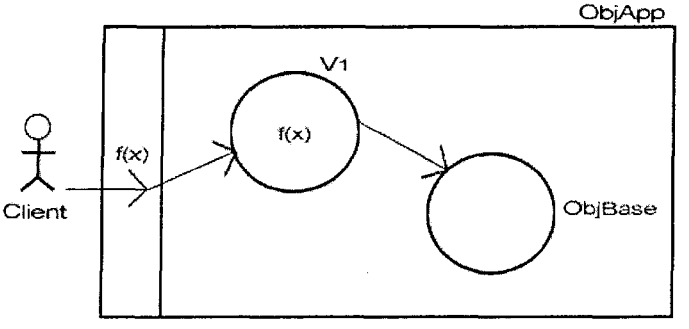

Figure 4.4 : ObjApp à t_1

Qui décide que le système de vues doit se transformer pour répondre aux mêmes besoins en fonction du temps?

Le client? Comment fait-il pour connaître les séquences d'appel interne de ObjApp? Doit-on fournir un guide détaillé à tous les clients? Ce n'est pas très convivial ni prudent... Après considération, le client désire seulement appeler f(x) sans se tracasser avec ce qui se passe à l'intérieur de la classe.

Mais quelle serait l'alternative? Automatiser cette évolution au cœur de ObjApp? Cette solution apporte aussi des lots de difficultés et nous amène vers une nouvelle problématique : « **L'évolution des besoins est complètement dans les mains du client et le système de vue est tributaire et sans réel contrôle** ». Le danger est de trop centraliser les décisions et de risquer de frustrer le client qui ne comprend plus trop ce qui se passe lorsqu'il appelle f(x). Où trancher?

4.2.6 Problème de droits

Soit ObjApp ayant les vues V1, V2 et V3. Un client C1 peut activer, désactiver, attacher ou encore détacher les vues V1, V2 et V3. Un client C2 peut faire exactement les mêmes opérations.

Il est impossible de dissimuler les fonctionnalités de l'objet, car tout client a accès à toutes les vues. De ce fait, on dénote que cette multitude de possibilités peut provoquer des conflits, mais surtout la perte de sens des vues. Le modèle de sécurité doit obligatoirement limiter l'accès à ces vues au client. Toute forme de transgression de cette limitation invaliderait la mise en place de la sécurité.

La problématique qui s'en dégage est la suivante : « **Tous les clients peuvent gérer toutes les vues de l'objet d'application, sans restriction ni contrôle!**»

4.2.7 Gestion des appels

Supposons un objet d'application ObjApp unique et partagé par tous les clients. Il contient cinq vues qui contiennent chacune une méthode f(x). Supposons aussi deux clients

C1 et C2 identifiés par un modèle de sécurité. C1 possède des accès à V1 et V3 et C2 possède des accès à V2 et V4.

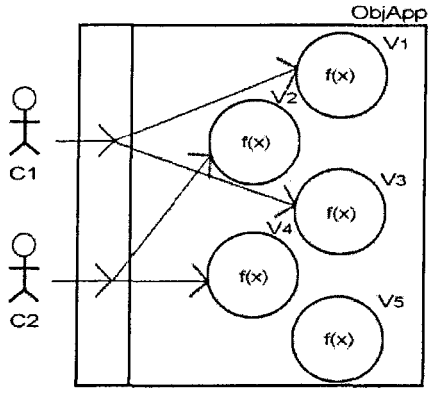

Figure 4.5 : Problématique découlant de la multitude des combinaisons (Cx, Vx)

Dans ce cas, la problématique illustrée à la figure 4.5 est plutôt un questionnement : « **Comment écrire le code de f(x) dans ObjApp en tenant compte du nombre élevé de possibilités découlant des combinaisons (Cx, Vx) (où Cx est un client quelconque et Vx une vue quelconque)** »

Exprimons cette réalité à l'aide d'un pseudo-code :

```
SI ClientCourant = C1 ALORS
    SI V1 est attachée et active ALORS
        APPELER V1.f(x)
    SI V3 est attachée et active ALORS
        APPELER V3.f(x)
SINON SI ClientCourant = C2 ALORS
    SI V2 est attachée et active ALORS
        APPELER V2.f(x)
```

SI V4 est attachée et active ALORS

APPELER V4.f(x)

Que faire si on a dix clients et une possibilité de dix vues? On répète cet exercice douteux pour toutes les méthodes de ObjApp? Le problème se complique énormément si le retour de f(x) dépend des vues activées... Ce point sera approfondi avec la présentation des problématiques de délégation (voir section 4.3) et d'exécution partielle (voir section 4.6)

4.3 La problématique de délégation

Reprenons la description de la délégation des appels énoncée à la section 2.4.4. La délégation est un mécanisme qui permet de transmettre l'appel de toute méthode d'un objet à l'autre. Dans notre cas, on détermine comment fonctionneraient les séquences d'appel à l'intérieur de l'objet d'application. Afin de mieux définir la problématique, repartons d'un cas simple (voir figure 4.6).

Soit un objet de base ObjBase ayant les méthodes f(x) et g(x)

Soit une vue V1 ayant elle aussi les méthodes f(x) et g(x)

Soit h(x), une méthode qui appelle directement l'objet de base. On se rappelle que l'implémentation dans l'objet d'application n'est pas tenue de toujours utiliser les vues.

On suppose que si un client appelle f(x) sur l'objet d'application ObjApp, ce dernier redirige l'appel vers la vue active, soit V1. Au cours de l'exécution, la vue appelle ObjBase.f(x). Le fonctionnement de g(x) est sensiblement le même.

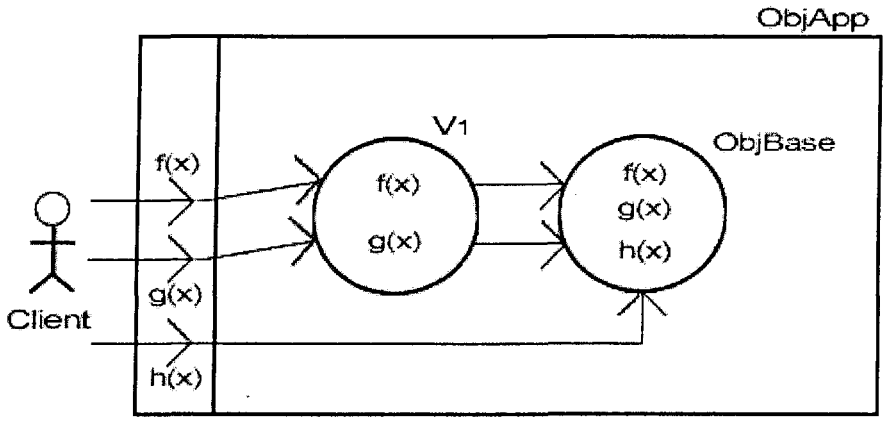

Figure 4.6 : Exemple de problématique de délégation

Ajoutons une variante : f(x) de la vue V1 doit faire appel à g(x). Quel g(x) sera utilisé? Nous avons trois possibilités :

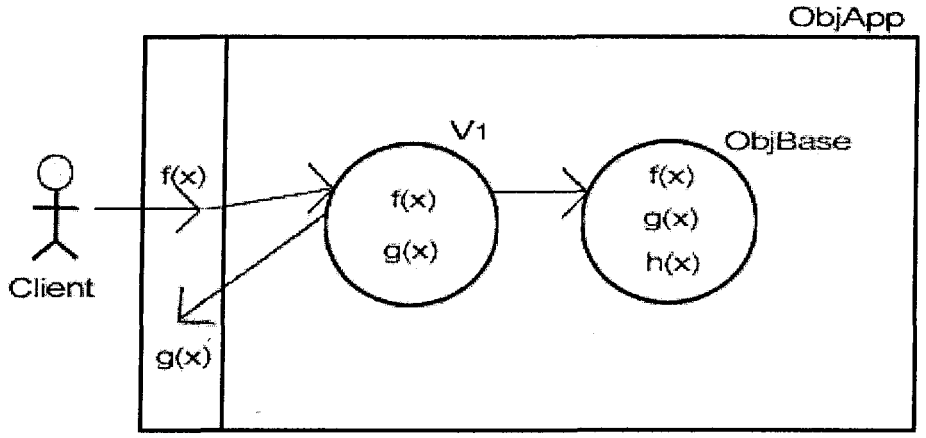

Figure 4.7 : La délégation respectée

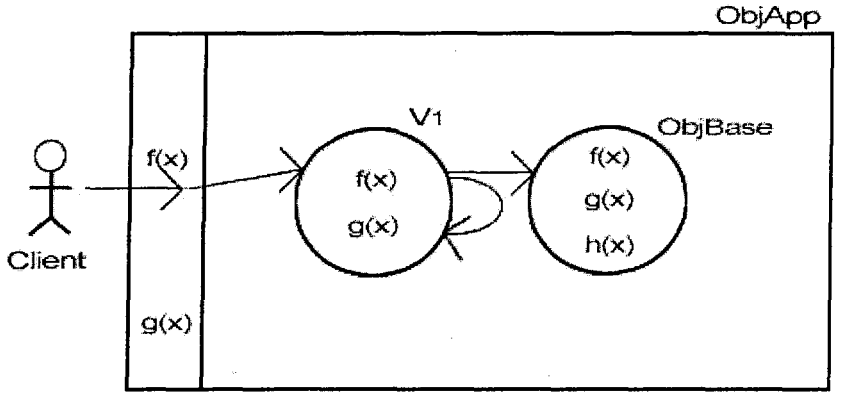

Figure 4.8 : La délégation brisée

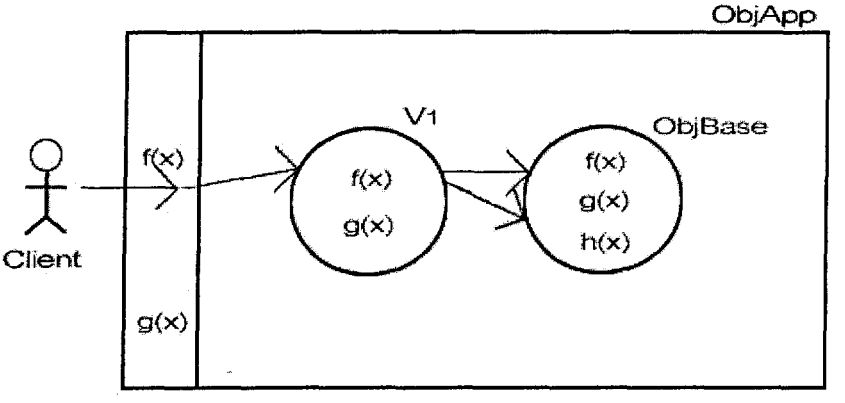

Figure 4.9 : délégation partielle

Dans le premier cas (voir figure 4.7), l'appel à la méthode g(x) de ObjApp semble peu adapté car la vue devrait conserver un lien vers ObjApp, ce qui ne sera pas le cas dans notre implémentation (voir figure 5.3). S'il y avait d'autres vues actives possédant elles aussi g(x), ObjApp devrait aussi leur faire appel. L'exécution pourra sortir du cadre fonctionnel défini par V1.f(x) ce qui n'est définitivement pas ce que l'on désire implémenter.

Lorsque la vue utilise sa propre méthode g(x), on nomme cette méthode délégation brisée (figure 4.8). Il s'agit en fait de la plus sécuritaire des options car on garde l'exécution à l'interne de la vue.

Le troisième cas (voir figure 4.9), où on ignore la vue et on file directement vers ObjBase se nomme délégation partielle et est considérée non sécuritaire. ObjBase est accédé par la vue passant outre les besoins fonctionnels définis par sa propre méthode g(x). En un sens, on défie la promesse d'encapsulation fonctionnelle promise avec le point de vue.

Il est clair qu'une seule approche doit être utilisée lors du choix de l'implantation de notre modèle de sécurité. Avec le nouveau modèle, nous ajoutons une gestion multiclient qui aura possiblement une influence sur le choix d'une ou de l'autre des types de délégation.

4.4 La problématique de sécurité

De loin la problématique la plus importante, la sécurité de l'objet d'application soulève une multitude de tracas.

Qui est le client? Il devrait être possible de rendre tout utilisateur de l'objet d'application identifiable via une authentification usager/mot de passe, une signature numérique, un objet ou tout autre procédé fiable. Si un client ne peut être identifié sans ambiguïté, que vaut un système sécurisé... Un objet du système devra avoir la responsabilité de fournir cette identification.

Qui gère les clients? Il sera nécessaire d'intégrer un mécanisme de validation de l'identité afin de permettre à ObjApp d'être tout à fait certain de l'identité du client avec lequel il discute.

Qui fait quoi? L'accès aux vues doit être contrôlé afin d'éviter que tout client identifié puisse utiliser toutes les vues actives. Sans gestion des droits et permissions, on ne fait que restreindre l'accès à l'objet et on ne règle pas le problème. En fait, on déplace la gestion de la sécurité à l'intérieur de l'objet d'application.

Où se situe exactement l'information à sécuriser? Doit-on superviser l'accès aux vues, aux méthodes des vues ou encore à l'objet de base lui-même? À quel niveau érige-t-on cette digue…

Finalement, comment planifier l'établissement d'un modèle de sécurité sans alourdir l'objet d'application de code de gestion ou introduire dans ObjApp des informations sur le profil des clients eux-mêmes. Comme nous l'avons vu au chapitre 3, on tente de résoudre cette problématique à l'aide du modèle de sécurité de Java.

4.5 La problématique de la synchronisation

Qui dit multi clients, dits problématique de synchronisation. À l'instant où l'appel d'une méthode influence l'état d'un objet partagé (vue, ObjApp ou ObjBase), on risque un conflit. Illustrons un cas général faisant ressortir la problématique de synchronisation.

Soit un client C1 qui appelle f(x) de la vue V1.

Soit un client C2 qui appelle simultanément V2.g(x)

Ajoutons une méthode getObjet(...) à ObjBase qui retourne une référence vers un objet quelconque Obj. Cette méthode est appelée lors de l'exécution de V1.f(x) et V2.g(x)

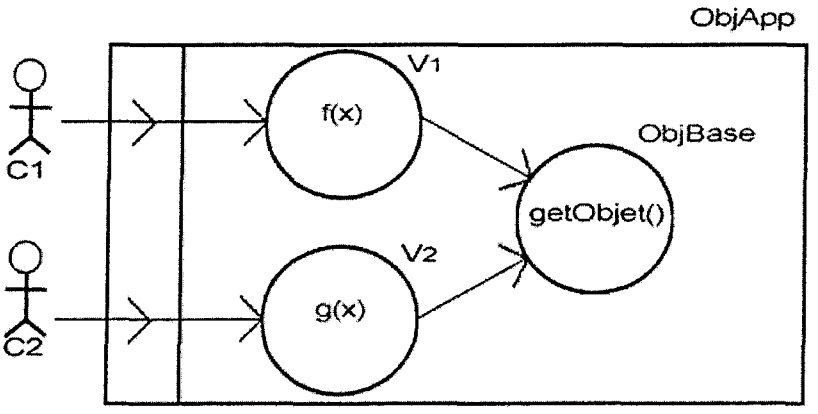

Figure 4.10 : La problématique de la synchronisation

L'exemple de la figure 4,10 montre que V1 et V2 partageant ObjBase et ils l'utilisent concurremment à leur bon gré. Les possibilités de problèmes engendrées par les opérations de f(x) et g(x) font à coup sûr frémir tout concepteur d'application aguerri.

L'utilisation de méthodes et d'objets synchronisés devrait aider à réduire l'impact de cette problématique mais dans ce cas, on relègue le tout au développeur de vues. En terminant, il faut rappeler que l'objet de base représente l'entité sur laquelle on veut apposer une couche fonctionnelle, une entité statique et peu évolutive. Toute modification directe de ObjBase devant être évitée, cela entrave dramatiquement l'ajout de synchronisation...

4.6 La problématique de l'exécution partielle

Par exécution partielle, il est entendu l'exécution d'une séquence de méthodes qui est interrompue soit par manque de permissions soit pour toute autre raison (ex : une exception). Utilisons de nouveau un exemple :

Soit trois vues V1, V2 et V3 ayant chacune une méthode f(x).

Supposons f(x) de ObjApp, une méthode qui retourne un résultat booléen et qui possède la logique suivante :

ObjApp.f(x)

Retourne V1.f(x) ET V2.f(x) ET V3.f(x)

Donc la méthode ObjApp.f(x) retourne vraie si et seulement si les trois vues retournent elles aussi vraies. Ajoutons un grain de sable dans le mécanisme, chaque appel de f(x) des vues modifie ObjBase.

Que se passe-t-il si le client C1 n'a pas accès à V3? On exécute V1.f(x), puis V2.f(x) et on plante dans V3.f(x) par manque de permissions? Tout se passerait sans problème n'étant que ObjBase a été modifié... Alors, on pourrait émettre la suggestion suivante : L'accès aux vues doit être validé avant l'exécution. Soit :

SI C1 possède la permission d'exécuter V1.f(x) ET V2.f(x) ET v3.f(x) ALORS

Exécuter le code

SINON que faire?

L'implémentation de la solution tentera de fournir les réponses à cette problématique. Il restera encore à prévoir l'imprévisible mais sur ce point, quel système est vraiment parfait?

4.7 En conclusion

Ce chapitre pose les questions nécessaires afin de faire ressortir les principaux défis identifiés lors de notre analyse et de nos tests. Certaines de ces problématiques pourront être résolues. Sur certaines, qui nous semblaient aussi sérieuses, nous aimerions tout de même offrir une réflexion.

Des solutions aux problématiques de synchronisation et d'exécution partielle sont offertes d'office dans la version distribué de Hamid Mcheick [Mcheick, 06]. Cette solution non distribuée propose tout de même quelques pistes de solution différentes.

Le chapitre suivant pose les bases de la solution du modèle de sécurité afin d'en comprendre les principes directeurs. Nous attendrons au chapitre 6 pour revoir le questionnement que nous venions de développer et ainsi valider les problématiques résolues par notre solution.

5

LE MODÈLE DE SÉCURITÉ DES VUES

Dans ce chapitre, un modèle de sécurité basé sur l'architecture JAAS de Java sera proposé et appliqué à un exemple concret de programmation par vue. JAAS a été abordé au chapitre 3 et les divers éléments de la programmation par vue se retrouvent au chapitre 2.

La présentation de la solution sera développée en trois sections. Une première section composée d'un côté schématique où sera progressivement construite une ébauche UML de la solution proposée et d'un cheminement complet illustré par des exemples d'exécution et des extraits de code. Ensuite, la seconde section concerne l'organisation de la solution Java : le code, les fichiers de configuration, l'implémentation et les possibilités d'évolution. Finalement, le chapitre 6 complète la solution en apportant réponse aux diverses propositions de problématiques définies au cours du chapitre précédent.

5.1 Analyse de la solution

Notre solution exploite le même exemple qui a été utilisé au chapitre 2 (voir figure 2.10 et annexe A). Établissons toutefois quelques éléments avant de débuter (illustrés à la figure 5.1) :

- Notre objet de base est représenté par la classe InfoEtudiant qui est elle-même liée à la classe Cours.
- La classe InfoEtudiantApp hérite de ObjApp. Elle est donc l'implémentation de l'objet d'application qui sera utilisée dans cet exemple.
- L'objet d'application parent ObjApp possède l'ensemble des vues implémentées (via un vecteur)
- Chacune des trois vues utilisées (VueFinance, VueAdministration et VueArchive) implémentent leur point de vue respectif mais partagent toutes le même parent VueInfoEtudiant.
- La classe VueInfoEtudiant hérite de la classe abstraite Vue, utilisée par l'objet d'application. Son unique rôle est de retourner l'objet de base et l'objet d'application avec le bon type. Son
- Le client a un lien direct avec InfoEtudiantApp par l'intermédiaire de la classe InfoEtudiantAppSecure qui assure la communication à travers sa classe parente ObjAppSecure

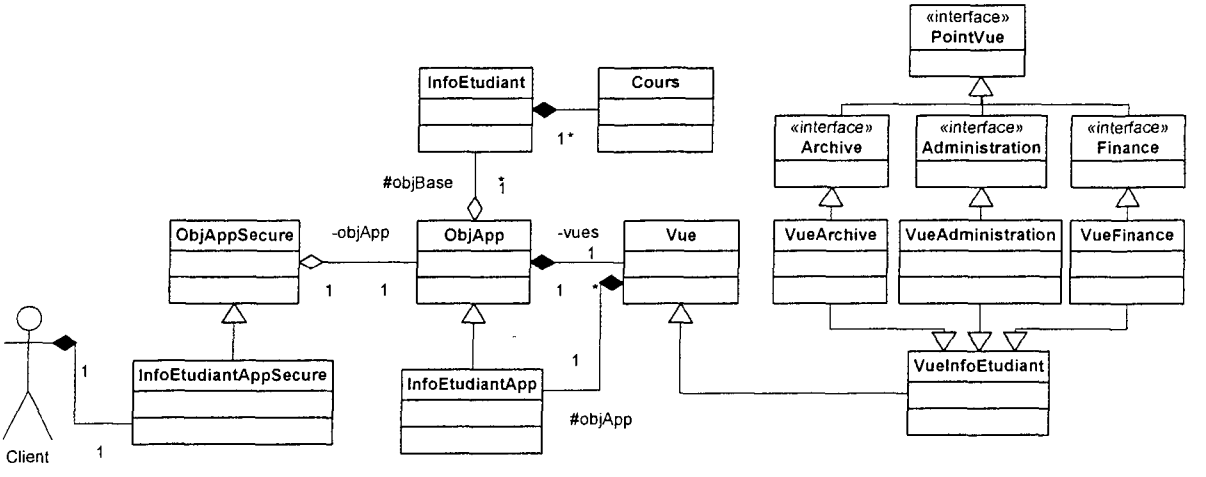

Figure 5.1 : Implémentation de la solution de départ

Avec ces quelques éléments, on situe le contexte de départ de l'approche. Un client utilise une classe de transition (InfoEtudiantAppSecure) afin de communiquer avec un objet d'application (ObjApp) qui a pour rôle de rediriger les appels aux vues et à l'objet de base (InfoEtudiant). Notre interface doit permettre une communication simple et transparente entre les deux parties. Rappelons-en une dernière fois les énoncés :

1. Le client doit être identifiable en tout temps
2. Le client doit être identifié par le système (ObjApp), aisément et sans ambigüité
3. L'utilisation du système doit être sécuritaire, transparent et personnalisé

5.1.1 Schéma global de l'object conceptuel ObjApp

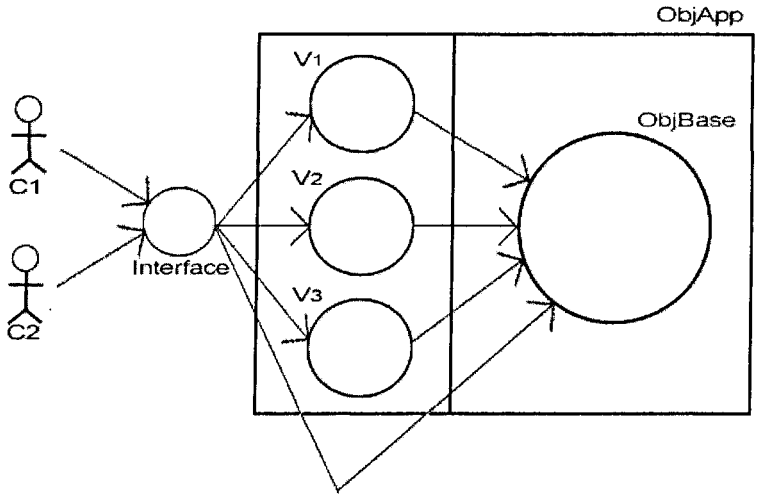

Figure 5.2 : Schéma global de l'objet conceptuel ObjApp avec interface

La figure 5.2 schématise l'approche globale qui définit l'ordre de présentation des sections du présent chapitre.

La section 5.2 explique en détail l'identification du client. Il faut comprendre comment deux clients anonymes (C1 et C2) peuvent utiliser une interface similaire et être reconnus comme des entités différentes.

La section 5.3 discute de la communication entre le client authentifié, où qu'il soit, et l'objet d'application sécurisé, où qu'il soit.

La section 5.4 expose les changements apportés aux vues afin de les rendre sécuritaires. Comment une vue V1, V2 ou V3 peut être identifiée comme accessible ou non pour un client Cx?

La section 5.5 complète le modèle de sécurité en décrivant les manipulations nécessaires permettant à l'objet d'application d'établir un lien entre le client identifiable et les vues sécurisées.

La section 5.6 discute de l'organisation et de l'implémentation en Java.

La section 5.7 suggère une piste de réflexion pour l'utilisation des aspects.

La section 5.8 complète en avec de nouveaux schémas UML, une vue plus complète de la solution.

5.2 Identification du client

Le client n'est en fait qu'un objet anonyme dans un système tout aussi anonyme. Il s'agit d'une généralisation de tout ce qui pourrait avoir besoin d'accéder aux ressources d'un autre système. Dans notre cas présent, cet autre système est représenté par l'instance de InfoEtudiantApp, classe enfant de ObjApp.

De ce fait, il est hors de question d'intégrer un processus d'authentification complexe qui devrait être implémenté partout, chez tous les clients. De plus, n'ayant aucun contrôle sur le fonctionnement de ses clients, l'objet d'application ne pourrait être certain de la véracité et de la légalité des appels provenant de l'extérieur.

L'objet d'application serait donc le second choix logique pour l'ajout de fonctions de sécurité. Le problème c'est qu'il ne peut savoir avec qui il discute à moins, et à chaque fois, de demander une preuve d'identité, ce qui est aussi impensable. Par exemple, la signature de la méthode enDette(...) de InfoEtudiantApp serait modifiée pour

enDette(ClientId id, ...). Par souci de transparence et de facilité d'utilisation, la signature des méthodes ne doit pas être modifiée.

Après réflexion, nous en sommes convenus que la seule solution potentiellement acceptable dans ce cas, serait d'intégrer une tierce partie au processus. Un serveur d'authentification régulariserait les échanges. On passe d'une relation entre deux parties <Client, ObjApp> vers une relation tripartie <Client, <ServeurAuthentification, ObjApp>>. Toutefois, le réel but de notre exercice étant l'utilisation simple d'un objet simple avec une sécurité transparente et autant simple, il nous est apparu que la tierce partie était en fait une solution un peu trop lourde, un canon pour tuer une mouche. De plus, il serait préférable de limiter la configuration, d'éviter les installations et de conserver le domaine d'exécution aux objets précédemment définis.

Pour répondre à notre besoin, nous utiliserons finalement JAAS (voir chapitre 3) qui offre un processus d'authentification et d'autorisation fiable, simple et facile à implanter.

5.2.1 La classe VOPLogin

Dans notre implémentation, la classe VOPLogin est un Singleton qui fournit au client une seule méthode login (String objBase) : ObjAppSecure.

Cette classe encapsule tout ce qui a trait au fonctionnement de l'authentification du client. Voyons son fonctionnement :

- Elle reçoit la demande du client sous la forme du nom de l'objet de base auquel on veut se rattacher
- Elle enclenche le processus d'authentification

- Si le processus est réalisé avec succès, cette classe crée et retourne l'entité du client, une instance de la classe Subject, bien encapsulée dans un conteneur de type ObjAppSecure (voir la section 5.2.3)

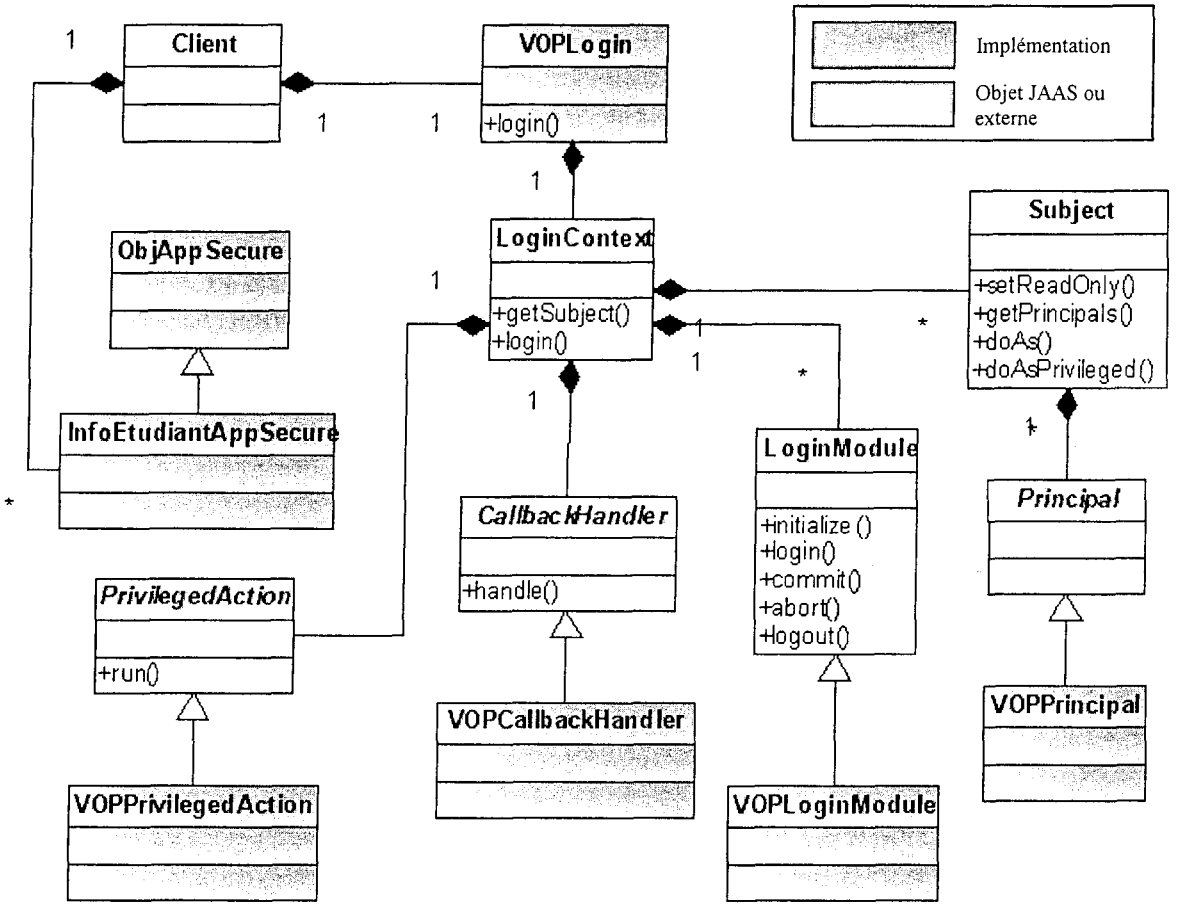

Figure 5.3 : Implémentation et encapsulation des classes reliées à l'authentification

La figure 5.3 complète celle introduite à la section 3.6. Notre implémentation y ajoute la classe VOPLoginModule qui s'exécute dès le début de l'authentification en réalisant une recherche d'informations sur le client. Un petit fichier de configuration (vop_jaas.config)

fournit au contexte d'authentification de JAAS la liste des modules à utiliser, dans ce cas, il ne contient que la ligne

Fichier de configuration vop_jaas.config

```
VOP {
   vop.securite.login.VOPLoginModule required;
};
```

Étant obligatoire (mot clé *required*), la réussite de l'authentification par le VOPLogin est une condition sine qua none pour la réussite de l'authentification du client.

Afin de limiter les modifications au code client, nous utilisons des *callback*, de petites classes qui permettent de gérer les flux d'entrée et de sortie de la communication. Notre classe VOPCallbackHandler permet au VOPLoginModule de communiquer directement avec le client via la console par défaut. Cette situation peut être considérablement éclaircie avec la capture d'écran de la figure 5.4 résultant de l'exécution du code qui suit immédiatement après.

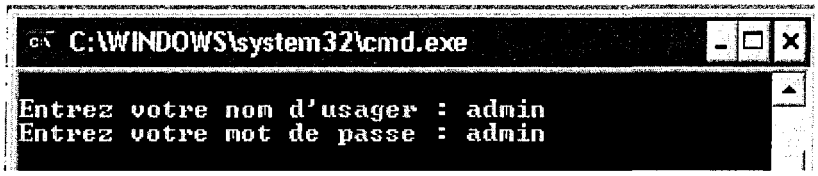

Figure 5.4 : Exécution du VOPCallbackHandler en console

Le code de la classe Client

```
/* CLIENT :    Connexion à l'objet d'application
               Ici le client appelle la méthode login de VOPLogin pour recevoir
               une instance de InfoEtudiantAppSecure encapsulant InfoEtudiantApp
*/

InfoEtudiantAppSecure infoEtudiantAppSecure = null;
```

```
try{
        infoEtudiantAppSecure =
        (InfoEtudiantAppSecure)VOPLogin.login("InfoEtudiant");
}
catch(VOPException vope)
{
        vope.printStackTrace();
        return;
}
...
```

Le code de la classe VOPLogin

```
/* VOPLogin :  Extrait du code de la méthode login appelée par le client
               Création du contexte, authentification et création de l'entité
               Lors de l'appel de la méthode login() de l'instance loginContext,
               la méthode login() de tous les LoginModule est appelée.
               Dans notre cas, nous avons un seul module, VOPLoginModule
*/
...
loginContext = new LoginContext("VOP", new VOPCallbackHandler());
loginContext.login();
Subject subject = loginContext.getSubject();
...
```

Le code de la classe VOPLoginModule

```
/* VOPLoginModule :   Lors de la création du contexte, le VOPLoginModule est instancié.
                      L'appel de la méthode login du contexte déclenche l'appel de
                      chaque méthode login de chaque module.
                      Communication avec le client pour authentification à l'aide des
                      Callback.  Après avoir récupéré les réponses, le loginModule décide
                      si oui ou non, l'usager est authentifié.
*/
...
Callback[] callbacks = new Callback[2];
callbacks[0] = new NameCallback("Entrez votre nom d'usager : ");
callbacks[1] = new PasswordCallback("Entrez votre mot de passe : ", false);

callbackHandler.handle(callbacks);
// Étapes de validation suite au retour des réponses du client
...
```

La suite des opérations du VOPLoginModule consiste à effectuer la vérification des informations fournies et à valider l'authentification. Pour plus de détails, voir le diagramme de séquences de l'authentification à la figure 5.6.

Dans l'implémentation actuelle, nous procédons simplement mais il serait envisageable de la remplacer par une authentification par LDAP (*Lightweight Directory Access Protocol* [IETF, 97]), par exemple, sans aucun impact sur le fonctionnement.

5.2.2 La classe Subject

La création de l'entité Subject s'effectue lors d'une authentification complétée. Cet objet sera utilisé pour rendre le client authentifiable. Il pourra être questionné par l'objet d'application et ainsi fournir l'identité VOPPrincipal qui ne contiendra qu'une seule information : l'identité du profil du client qui aura été ajouté par le VOPLoginModule lors de l'appel de la méthode commit(...) (voir section 3.4.3, phase 3)

5.2.3 La classe ObjAppSecure

L'authentification à travers VOPLogin retourne une instance de ObjAppSecure sous la forme d'une classe enfant, dans notre cas, InfoEtudiantAppSecure. ObjAppSecure servira de passerelle vers l'objet d'application auquel elle possède maintenant une référence, fournie par VOPLogin. ObjAppSecure encapsule aussi l'entité, l'instance de la classe Subject, représentation sécurisée du client. Il est à noter que le client n'a pas accès aux classes Subject et InfoEtudiantApp étant toutes deux des objets privés.

Le client communique avec sa propre instance de InfoEtudiantSecure qui se charge de transmettre sa demande à ObjAppSecure (sa super-classe) à l'aide de la méthode execute(...), d'attendre le retour et à retransmettre au client le résultat de l'appel.

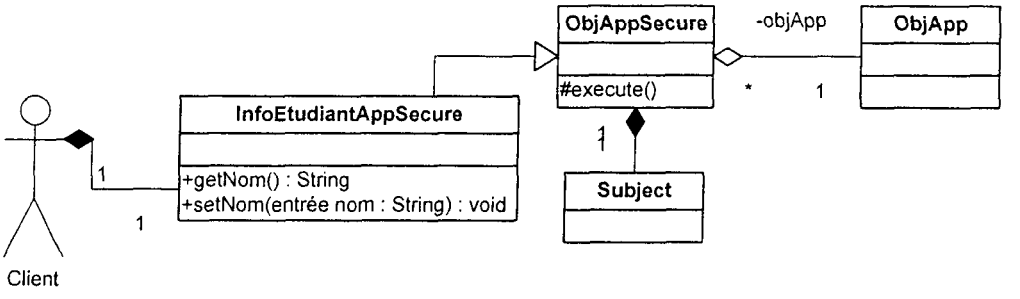

Figure 5.5: Les classes de InfoEtudiantAppSecure

Un extrait du code des appels des méthodes setNom(...) et getNom(...) se retrouve ci-dessous

L'appel par le client des méthodes setNom(...) et getNom(...) de la classe Client

```
...
System.out.print("* Entrer un nouveau nom    : ");
infoEtudiantAppSecure.setNom((new BufferedReader
(new InputStreamReader(System.in))).readLine());
System.out.println("* Le nom est : " + infoEtudiantAppSecure.getNom());
...
```

Les méthodes setNom(...) et getNom(...) de la classe InfoEtudiantAppSecure

```
...
public void setNom(String nom) throws VOPException{
        Class<?> params = String.class;
        Object[] valeurs = {nom};
        execute("setNom", params, valeurs);
}

public String getNom() throws VOPException{
        return (String)execute("getNom");
}
...
```

La classe InfoEtudiantAppSecure n'a aucun état ni autre fonctionnalité. Elle pourrait donc être aisément générée d'une manière ou d'une autre. Le fonctionnement de la solution repose principalement sur une petite classe charnière, la classe abstraite ObjAppSecure et son unique méthode execute(...). Cette manipulation semble complexe,

mais elle permet d'éviter une transformation de la signature de la méthode du côté client et du côté des vues, ce qui élimine la majorité des modifications.

La méthode execute(...) utilise le *reflection package* de Java qui offre la possibilité de décomposer un appel de méthode en objet. Ceci fait, ObjAppSecure communique avec l'objet d'application et lui envoie l'objet représentant la méthode à exécuter, la signature à utiliser et les paramètres fournis par le client.

Nous avons finalement une solution côté client raisonnablement transparente n'ayant aucune installation ou configuration significative. Il suffit d'ajouter le fichier contenant les politiques de sécurité à la solution Java du client. Ce fichier est nécessaire pour permettre au programme en exécution d'avoir accès au domaine de sécurité. Pour plus d'informations, revoir la section 3.5.2 et consulter la prochaine section 5.4.2 qui exposera plus en détail le fichier de notre exemple.

5.2.4 Diagramme de séquence de l'authentification

Considérant la quantité de classes en jeu et la difficulté de bien assimiler la solution à l'aide de simples classes, un diagramme de séquences résumant les diverses étapes de l'authentification est présenté en guise de conclusion à la figure 5.6. (rappel : La solution Java complète se retrouve à l'annexe B.)

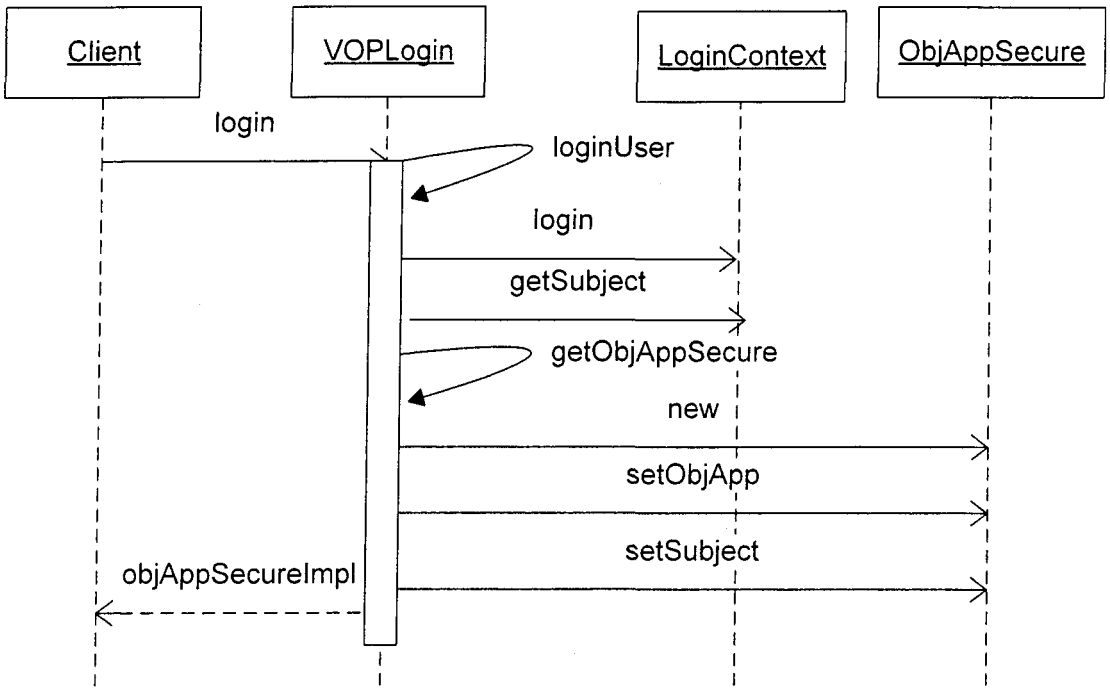

Figure 5.6 : diagramme de séquence de l'authentification

5.3 La communication

La communication entre le client, l'objet ObjAppSecure, et ObjApp peut être réalisée de plusieurs manières. Dans l'optique de démontrer le principe de sécurité, nous avons limité l'investissement en analyse et en développement de ce côté. La relation proposée est très simple : la classe client possède une relation de composition vers l'objet d'application, dans la même exécution et sur le même ordinateur.

Ceci étant dit, des travaux ont déjà été réalisés sur la distribution d'objets avec les techniques de développement orientées aspects et pourraient être évalués pour combler cette lacune [Mcheick, 06].

5.4 Les vues

Les vues se définissent comme un ensemble de filtres fonctionnels liés à un objet de base immuable. Dans notre approche initiale, les vues étaient partagées par tous les utilisateurs du système et il y avait une seule instance de chacune (À noter : certaines implémentations de la programmation par vue proposent l'instanciation multiples des vues). Notre solution conserve l'unicité des vues, seulement la notion de partage est modifiée. En fait, on utilisera une simple règle implémentée dans ObjApp.

Algorithme pour attacher et sécuriser une vue

Si on veut accéder à une vue

Si la vue n'est pas attachée à l'objet d'application

On identifie la vue

On instancie une permission d'accès à cette vue

On instancie la vue et on la lie à la permission

La vue est sécurisée et maintenant accessible

Retourner la vue à l'instance de l'objet d'application

Voici donc le code correspondant de la méthode attacherVue(...) de la classe ObjApp

La méthode attacherVue(...)

```
private void attacherVue(String vue) throws VOPException{

        try{
                String nomClasse = objBase.getClass().getName();
                int debutNom = nomClasse.lastIndexOf(".") + 1;
                nomClasse = nomClasse.substring(debutNom);
                nomClasse = "vopimpl." + nomClasse.toLowerCase() + ".vue.Vue" + vue;
                Class nouvelleClasse = Class.forName(nomClasse);
                Constructor[] constructors = nouvelleClasse.getConstructors();
```

```
                VuePermission perm = new VuePermission(vue);
                Vue nouvelleVue = (Vue)constructors[0].newInstance(objBase, this, perm);
                vues.put(vue, nouvelleVue);
        }
        catch (Exception e){
                String message  =  "Impossible d'attacher la vue " + vue +
                                   " (" + e.getMessage() + ")";
                throw new VOPException(message);
        }
}
```

Des problèmes de synchronisation, notamment liés au partage de vue possédant un état (des attributs), sont possibles, mais l'intégration d'un mécanisme de gestion (tel un StatefulBean des EJB de Java) alourdirait inutilement le code et ne serait d'aucun secours à notre solution. À l'origine, il était possible d'activer et de désactiver des vues préalablement attachées afin de simuler un profil particulier. Avec cette solution, il n'est plus possible de le faire manuellement si on considère à nouveau notre client comme un simple utilisateur et non plus comme un gestionnaire de vues, responsable de configurer l'objet d'application pour rendre ses comportements correspondant au besoin.

5.4.1 La classe VuePermission

La gestion des autorisations tourne autour d'une classe nommée VuePermission (voir section 3.5.1 pour plus de détails sur les permissions et la figure 5.7 pour la représentation de l'héritage). Elle contient le nom de la vue à sécuriser, reçu en paramètre du constructeur. Cette information sera utilisée par le gestionnaire de sécurité pour valider les accès à la vue elle-même. Il demandera à la vue « Qui es-tu? Je dois te reconnaître pour te permettre de continuer » et cette dernière répondra «On peut me reconnaître par la valeur contenue dans ma permission ». Voir la section 5.5.5 pour le processus plus complet.

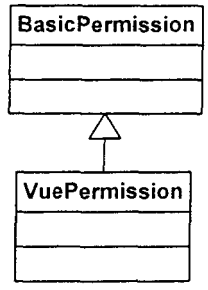

Figure 5.7 : Héritage de VuePermission

Dans notre solution, les possibles exceptions, instances de VOPException, seront retournées au client via la pile d'appel (*call stack*). En fait, tout appel du client devra être réalisé à l'intérieur d'un bloc de gestion d'exception (try...catch) au cas où celui-ci ferait appel à une méthode contenue dans une vue où il n'a pas les permissions nécessaires.

Exemple de code client avec gestion d'exceptions

```
...
        try{
            System.out.println("* La balance est de : " +
            infoEtudiantAppSecure.getBalance());
        }
        catch(VOPException vope){
            vope.printStackTrace();
        }
...
```

5.4.2 Configuration des vues

Les permissions d'accès aux vues sont configurées dans un fichier de politiques de sécurité qui est administré par le gestionnaire de sécurité (voir section 3.5.2). Jetons un coup d'œil sur le fichier de configuration de notre exemple.

Le fichier vop.policy

```
/*****************************************************************************
 *** SECTION #1 : Gestion des ressources systèmes                           ***
 *****************************************************************************/
grant codebase "file:./vop.jar" {
       /* Gestion des permissions particulières pour le LoginModule */
       permission javax.security.auth.AuthPermission
```

```
                "modifyPrincipals";
        permission javax.security.auth.AuthPermission
                "createLoginContext.VOP";
        permission javax.security.auth.AuthPermission "doAsPrivileged";
};
grant codebase "file:client.jar" {
        permission javax.security.auth.AuthPermission
                "createLoginContext.VOP";
        permission javax.security.auth.AuthPermission "doAsPrivileged";
};
grant codebase "file:vopimpl.jar" {
        permission javax.security.auth.AuthPermission "doAsPrivileged";
};


/*****************************************************************************
 *** SECTION #2 : Gestion des permissions reliées aux vues                 ***
 *****************************************************************************/

/** Profil : ADMIN **/
grant   codebase "file:vop.jar", Principal vop.securite.VOPPrincipal
        "admin" {
        permission vop.securite.VuePermission "Administration";
        permission vop.securite.VuePermission "Archive";
};
grant   codebase "file:vopimpl.jar", Principal
        vop.securite.VOPPrincipal "admin" {
        permission vop.securite.VuePermission "Administration";
        permission vop.securite.VuePermission "Archive";
};

/** Profil : ARCHIVE **/
grant   codebase "file:vop.jar", Principal vop.securite.VOPPrincipal
        "archive" {
        permission vop.securite.VuePermission "Archive";
};
grant   codebase "file:vopimpl.jar", Principal
        vop.securite.VOPPrincipal "archive" {
   permission vop.securite.VuePermission "Archive";
};

/** Profil : INVITE **/
grant   codebase "file:vop.jar", Principal vop.securite.VOPPrincipal
        "invite" {
};
grant   codebase "file:vopimpl.jar", Principal
        vop.securite.VOPPrincipal "invite" {
};
```

La première section définie les droits d'accès aux divers objets système pour l'exécution des méthodes d'authentification (voir tableau 5.1 pour le contenu des fichiers d'archive). La seconde section concerne l'accès aux vues. Par exemple, on peut remarquer que l'identité *admin* de type VOPPrincipal a accès aux vues administration et archive et que l'identité *invite* est reconnu mais ne possède aucun accès.

L'ensemble de la personnalisation des accès aux vues est complètement réalisé à l'extérieur du code ce qui facilite son dynamisme. On peut ajouter et supprimer des accès aux vues, donc modifier les comportements de l'objet d'application, sans toucher au programme. Il ne reste seulement qu'à sécuriser le fichier sur le disque, ce qui est trivial.

5.4.3 Sécurité des vues

Reprenons le diagramme de classes de la figure 2.10 correspondant à l'implémentation de la solution sans module de sécurité et comparons-le avec ce que nous venons tout juste d'aborder. Vous pourrez valider les similitudes et changements avec les codes Java en annexe.

- Les points de vue restent des concepts abstraits et ne sont pas modifiés par les ajouts de sécurité
- L'implémentation des points de vues, c'est-à-dire les vues de InfoEtudiantApp, est développée pour répondre à un besoin fonctionnel qui doit, en tout temps, être indépendant des permissions de celui qui en fait appel. Dans notre implémentation, la seule modification se situe au niveau du constructeur qui reçoit une permission en paramètre et qui la transfère à sa classe parent, Vue.
- On ajoute la méthode protégée getPermission(...) à la classe abstraite Vue ce qui ne constitue pas un changement de vocation.

Mais où est donc l'implémentation de la sécurité? Toute la gestion de la sécurité se retrouve dans la classe ObjApp. Pourquoi? Il était primordial de bien conserver la séparation entre l'évolution des comportements et la sécurité. Même si, lors du

fonctionnement, les autorisations influencent largement sur l'utilisation des vues, les deux se devaient d'être mutuellement exclusives lors de la conception et du développement afin d'assurer un maximum de portabilité.

5.5 L'objet d'application

L'objet d'application conserve toutes les fonctionnalités qui lui ont été attribuées auparavant soit attacher/détacher une vue, fournir une vue et établir un lien vers l'objet de base. La principale différence consiste en l'ajout de deux méthodes critiques : getVue(...) et execute(...). On peut prendre connaissance des diverses méthodes à la figure 5.8

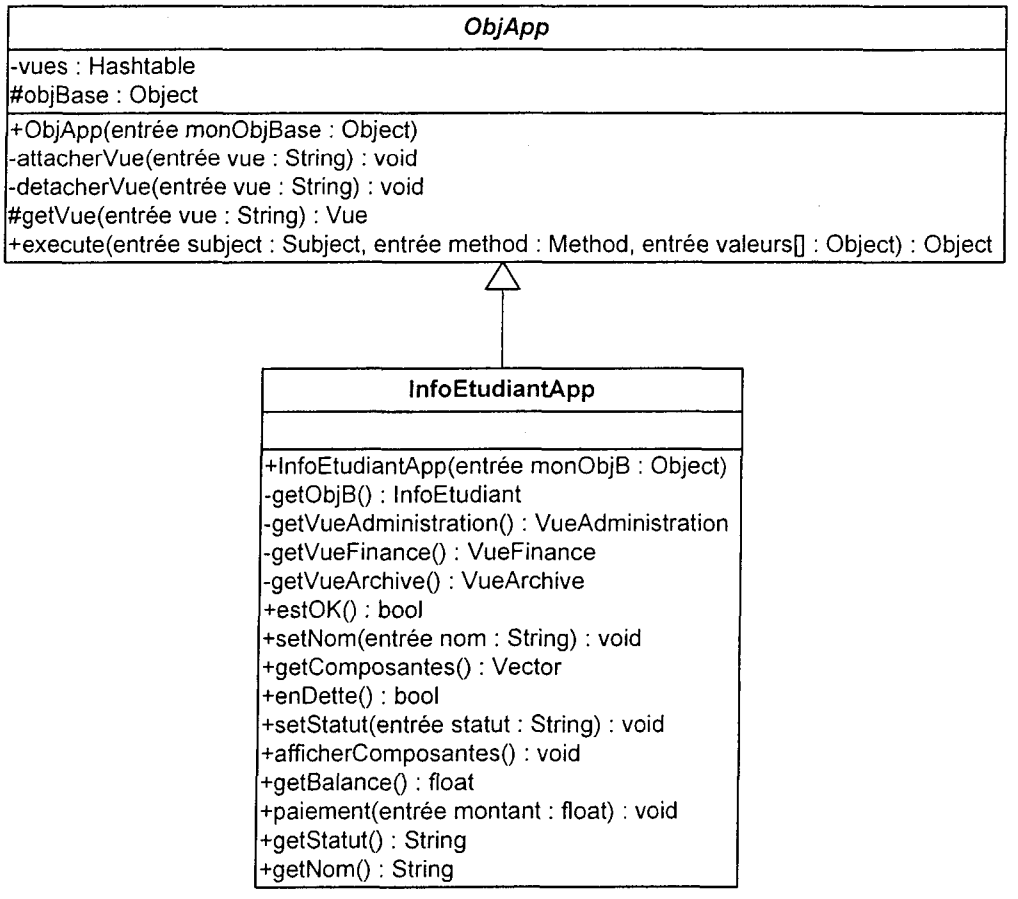

Figure 5.8 : Les classes ObjApp et InfoEtudiantApp

5.5.1 La méthode getVue(...)

Afin de sécuriser l'accès aux vues, on ajoute l'énoncé suivant :

« Tout accès à une vue doit être autorisé par la gestionnaire de sécurité. »

On retrouve dans le code source de la méthode getVue(...), l'implémentation de la logique de fonctionnement des permissions

La logique :

Trouver la vue active

Demander une référence vers la VuePermission associée à la vue (voir section 5.4.1)

Vérifier la permission (voir section 5.4.2)

Si la permission est valide

Retourner la vue

Sinon

Lancer une VOPException (voir section 5.5.2)

Le code de la méthode getVue(...)

```
final protected Vue getVue(String vue) throws VOPException {

        if (!vues.containsKey(vue))
                attacherVue(vue);
        Vue vueCourante = (Vue)vues.get(vue);
        VuePermission perm = vueCourante.getPermission();
        try{
                AccessController.checkPermission(perm);
        }
        catch(Exception e){
                String message  = "Accès interdit à la vue " + vue + " ("
                                + e.getMessage() + ")";
                throw new VOPException(message);
        }
        return vueCourante;
}
```

L'accès protégé et la définition finale de la méthode getVue(...) assure la protection des vues à l'intérieur de l'héritage. La méthode retourne une classe répondant aux promesses de l'interface InfoEtudiantApp (classe qui hérite elle-même de l'interface Vue). Par exemple, pour atteindre la vue VueAdministration, on utilise la méthode privée getVueAdministration(...) de InfoEtudiantApp qui a pour rôle de changer l'interface d'utilisation de l'objet retourné. La vue générique sera retournée en tant que VueAdministration.

Le code de la méthode getVueAdministration(...) de InfoEtudiantApp

```
private VueAdministration getVueAdministration() throws VOPException {
        return (VueAdministration)getVue("Administration");
}
```

Ce qui revient à dire que l'accès à l'ensemble des vues est protégé par l'objet d'application et c'est grâce à cette centralisation qu'il est possible de sécuriser tous les accès.

5.5.2 La méthode execute (...)

La méthode execute(...) relie l'objet ObjAppSecure du client à l'objet d'application. À l'aide d'une action sécurisée, une PrivilegedAction, on tente d'exécuter la méthode demandée par le client dans un contexte de sécurité relié à l'entité Subject. Il est très important de bien comprendre la différence entre le contexte de sécurité lié à l'exécution courante et celui qui est particulier à l'entité : le premier sans aucun privilège particulier, le second avec une identité VOPPrincipal.

Le code de la méthode execute(...)

```
public Object execute(Subject subject, Method method, Object[] valeurs) throws
VOPException{
        Object retour = Subject.doAsPrivileged(
                subject, new VOPPrivilegedAction(
                method, valeurs, this), null);
        if (retour instanceof VOPException)
                throw (VOPException)retour;
        return retour;
}
```

Réalisons l'analyse de cette méthode centrale en quatre blocs : la signature, l'instanciation de l'action privilégiée, son utilisation via la classe Subject et la gestion du retour.

Blocs	Analyse
La signature `public Object execute(Subject subject, Method method, Object[] valeurs) throws VOPException`	Les 3 paramètres reçus du client via la classe ObjAppSecure sont les suivants : subject : C'est l'entité représentant le client, il faut se rappeler qu'elle contient les identités du client qui est, dans notre solution, le profil à utiliser pour valider l'accès aux vues. Par exemple « admin » method : C'est la méthode, demandée par le client, qui devra être exécutée. Par exemple « setNom ». valeurs : Les valeurs à transmettre à la méthode. Par exemple « eric »

`new VOPPrivilegedAction(method, valeurs, this)`	L'action privilégiée est décrite à la section suivante. Pour le moment, on remarque qu'on lui transmet la fonction à appeler, ses paramètres et l'objet d'application courant (*this*) sans pour autant le connaître.
Utilisation de l'action privilégiée `Subject.doAsPrivileged`	doAsPrivileged est une méthode statique de la classe Subject. Son rôle est le suivant : réaliser l'exécution d'une action (PrivilegedAction) dans un contexte de sécurité particulier (celui de l'entité subject). À partir de ce moment, les identités de l'entité seront utilisées par le gestionnaire de sécurité.

Gestion du retour
`if (retour instanceof VOPException)`
`  throw (VOPException)retour;`
	L'appel de méthode retourne une instance très générique de type Object qui correspond à la valeur de retour de la méthode appelée par l'action privilégiée. Il n'est pas possible de contrôler directement la valeur de retour de la méthode doAsPrivileged(...). Nous retournons donc les exceptions d'accès (VOPException), telles que des variables de retour ce qui explique l'utilisation de l'opérateur de comparaison

	instanceof. La gestion est, pour ainsi dire, redirigée manuellement à ce point.

La méthode execute devient la seule porte d'entrée pour l'exécution de méthodes sécurisées dans l'objet d'application car il utilise le Subject qui transporte la signature du client nécessaire à l'autorisation (voir section 3.5.3)

5.5.3 La classe VOPPrivilegedAction

Il ne faut qu'une seule ligne de code dans VOPPrivilegedAction pour déclencher tout le processus d'autorisation, le reste n'étant que gestion secondaire

```
return objAPPMethod.invoke(objApp, valeurs);
```

En réutilisant le *reflection package* de Java, on effectue l'appel de méthode représentée par l'objet *objAppMethod*, sur l'objet *objApp* reçu en paramètre avec les paramètres *valeurs*. Grâce au polymorphisme, la bonne méthode est effectivement appelée dans la classe InfoEtudiantApp.

Par exemple,

objAPPMethod → getNom

objApp → infoEtudiantApp

valeurs → eric

est équivalent à

return infoEtudiantApp.getNom("eric");

à l'exception près qu'on exécute maintenant cet appel dans le contexte de l'entité.

5.5.4 Implémentation

Prenons un autre exemple, l'appel de la méthode estOK(...). La méthode execute(...) de ObjApp utilise une action privilégiée qui exécute la méthode estOK(...) de la classe InfoEtudiantApp qui en hérite. Dès que le processus a besoin d'une vue, ses accès sont validés à travers la méthode getVue(...) de l'objet d'application. Comment? Il faut garder à l'esprit que l'entité effectue cette opération dans son propre contexte de sécurité, ce qui revient à dire celui du client authentifié.

Comme pour tout programme, l'implémentation finale de chacune des méthodes devra être déterminée préalablement. Pour ce problème, la vue VueAdministration est obligatoire pour l'exécution de la méthode estOK(...) tandis que l'accès aux deux autres vues est optionnel. À part la logique fonctionnelle, le code est généralement assez simple. Certains patrons récurrents pourraient potentiellement y être découverts.

Le code de la méthode estOK(...) de InfoEtudiantApp

```
public boolean estOK() throws VOPException {

        /*      La vue administrateur est obligatoire mais les autres vues sont
                optionnelles.
                Un manque de permission à la vue Administrateur retourne une VOPException
                (section 5.5.1, méthode getVue(...).
                Autrement, on continue l'exécution.
        */

        boolean ok = getVueAdministration().estOK();

        try{
                ok &= getVueFinance().estOK();
        }
        catch (VOPException vope){
                System.out.println("OPTIONNEL : Acces interdit a la
                        vue Finance");
        }
        try{
                ok &= getVueArchive().estOK();
        }
        catch (VOPException vope){
                System.out.println("OPTIONNEL : Acces interdit a la
                        vue Archive");
        }
        return ok;
}
```

5.5.5 Diagramme de séquences de l'autorisation

Afin de bien synthétiser notre approche de permissions, le diagramme de séquence de la figure 5.9 reprenant les diverses étapes de l'autorisation est présenté en guide de conclusion.

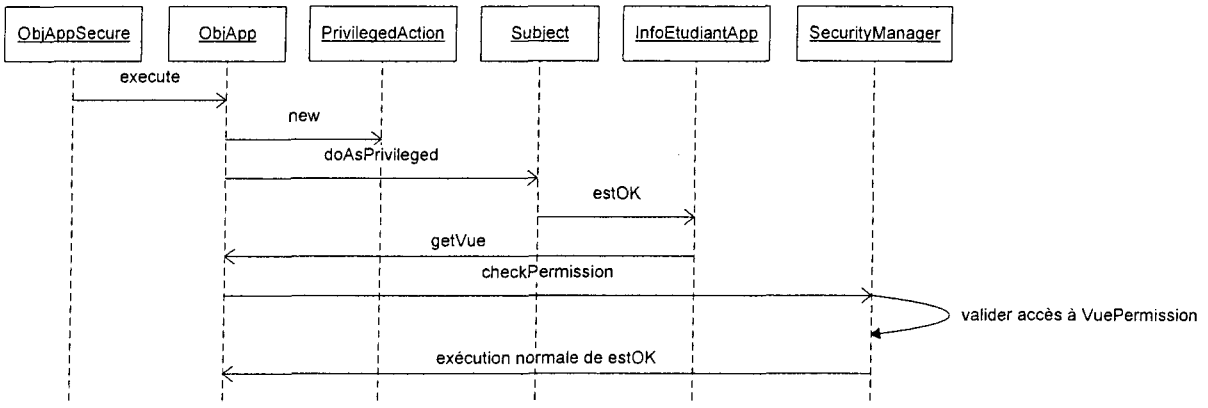

Figure 5.9 : Diagramme de séquences de l'autorisation

5.5.6 Exemples d'exécution d'une méthode sécurisée

Code du client qui appelle la méthode estOK(...). (La version complète du code se retrouve en annexe)

L'appel de la méthode estOK(...)

```
System.out.println("\n________\nTEST #1\n________");
try{
        boolean estOK = infoEtudiantAppSecure.estOK();
        System.out.println("* estOK    : " + estOK);
}
catch(VOPException vope)
{
        vope.printStackTrace();
}
```

La figure 5.10 permet de visualiser un exemple d'exécution avec profil « admin » (voir fichier de politique de sécurité à la section 5.4.2)

Figure 5.10 : Exemple d'exécution avec profil « admin »

La figure 5.11 permet de visualiser un exemple d'exécution avec profil « invite » (voir fichier de politique de sécurité à la section 5.4.2)

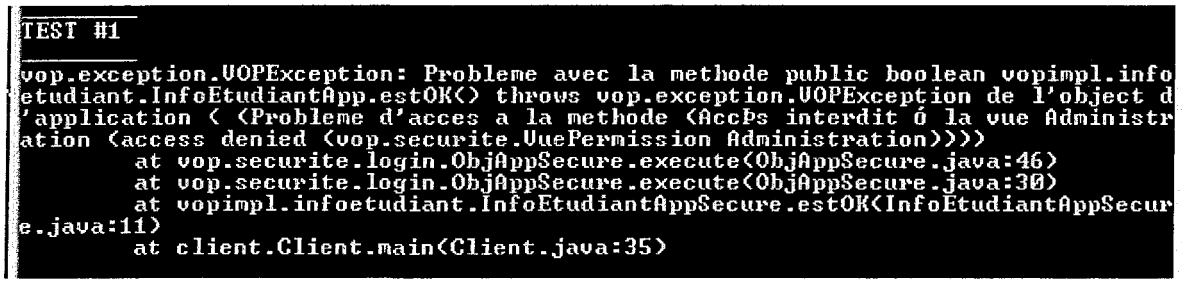

Figure 5.11 : Exemple d'exécution avec profil « invite »

5.6 La solution avec Java (JAAS)

Cette proposition de modèle de sécurité sur les vues, quoique plutôt abstraite à la base, repose en fait sur JAAS, donc sur Java. Il existe peut-être d'autres produits comparables sur le marché alors il ne semble pas pertinent d'étaler toutes les nécessités techniques reliées à l'installation et à la configuration. Toutefois, par souci de rendre notre solution acceptable, l'environnement technologique sera abordé à travers l'organisation logique de notre information et de notre code source. Ensuite, certaines contraintes et restrictions reliées à la programmation sont présentées.

5.6.1 Organisation

La version du JDK (Java Development Kit) est le 1.6 et l'environnement de développement est Netbeans 1.5. Il aurait pu en autre autrement avec des produits différents car notre implémentation est simple et portable. Afin d'améliorer la compréhension de la solution, les diverses classes de l'exemple ont été réparties à travers dix regroupements logiques distincts appelés, en Java, des *packages.*

Package	Description
Base	Contient l'ensemble des classes du système à être encapsulées par un objet d'application. On y retrouve les classes identifiées par le terme ObjBase tout au long du document Classes : *InfoEtudiant, Cours*
Client	Contient l'ensemble des clients anonymes utilisés pour réaliser des tests. Classe : *Client*
Vop	Contient toutes les classes de la programmation par vues communes à toutes les solutions. Elles ne sont jamais modifiées.
.app	Contient la classe représentant l'objet d'application ObjApp Classe : *ObjApp*
.exception	Contient les classes d'exception utilisées dans les processus Classe : *VOPException*
.securite	Contient l'ensemble des classes reliées à l'implémentation de

	l'architecture JAAS Classes : *VOPCallbackHandler, VOPPrincipal, VOPPrivilegedAction, VuePermission*
.securite.login	Contient les classes utilisées pour effectuer le suivi du processus d'authentification du client. Lors de l'authentification, ces classes utilisent des méthodes à accès restreint au package. Elles se regroupent afin d'améliorer la sécurité de l'objet Subject Classes : *ObjAppSecure, VOPLogin, VOPLoginModule*
.vue	Contient les classes utilisées pour définir les interfaces des vues et points de vue Classe : *Vue*
.vue.pointvue	Contient les classes de tous les points de vue, utilisés par l'ensemble des objets d'application, qui pourront être implémentées dans la solution Classes : *Administration, Archive, Finance, PointVue*
Vopimpl	Contient l'ensemble de la solution qui doit être développé pour chaque objet de base. Pour rendre un ObjBase utilisable, on doit créer un package qui porte exactement le même nom dans vopimpl. Reprenons l'exemple de l'objet de base InfoEtudiant
.infoetudiant	Contient les classes d'implémentation de ObjApp et ObjAppsecure Classes : *InfoEtudiantApp, InfoEtudiantAppSecure*

.infoetudiant.vue	Contient les classes qui implémentent les points de vue du package vop.vue.pointvue ainsi que la classe passerelle. Classes :*VueAdministration, VueArchive, VueFinance, VueInfoEtudiant*

Tableau 5.1 : Packages Java du projet

Cette organisation de packages permet de structurer l'information dans le but de générer quatre fichiers d'archive Java (fichier JAR) ayant chacun un rôle spécifique. Ces fichiers sont primordiaux car les permissions d'accès leurs sont attribuées. Par exemple, la permission d'ajouter des identités VOPPrincipal à l'entité Subject n'est possible qu'aux classes contenues dans l'archive vop.jar :

Le fichier vop.jar

```
grant codebase "file:./vop.jar" {
    permission javax.security.auth.AuthPermission
               "modifyPrincipals";
};
```

Archive	Description
base.jar	Ce fichier contient l'ensemble des classes représentant l'entité « objet de base ». Ce fichier devrait être assez statique et être modifié qu'avec l'ajout d'un nouvel objet de base
client.jar	Ce fichier contient les classes du « client ». Il ne serait pas nécessaire d'archiver ces classes mais c'est plus simple pour tester la solution.
vop.jar	Cet archive contient toutes les classes du package vop c'est-à-

	dire toutes les classes communes de la programmation par vues et l'aspect sécurité
vopimpl.jar	L'implémentation actuelle de la solution. Dans notre exemple, cette archive contiendra toutes les classes reliées à InfoEtudiant

Tableau 5.2 : Fichiers d'archive (jar) du projet

5.6.2 Implémentation

L'organisation des fichiers a été réalisée méthodiquement afin de s'assurer que l'implémentation de l'objet de base se réalise avec un minimum de développement. Le fichier d'archive vop.jar est complet en lui-même et reste constant d'une application à l'autre. C'est le cœur de la programmation par vue augmenté d'un fond de sécurité.

Partons d'un exemple avec objet de base et vues basées sur des points de vue. Même si les interfaces des points de vue sont identiques d'un objet de base à un autre, il n'en reste pas moins que chaque objet d'application a sa manière propre de répondre à chaque appel de méthode. Il en sera de même pour toutes les classes qui héritent de ObjApp (par exemple, InfoEtudiantApp). ObjApp est le proxy entre l'objet de base et les vues mais les détails de l'implémentation de InfoEtudiantApp doivent être écrits quelque part. Il serait possible d'envisager que la classe InfoEtudiantApp puisse être générée à partir d'un outil où l'on définirait les règles fonctionnelles de son utilisation.

Tout l'aspect de la sécurité est encapsulé à l'intérieur des classes ObjApp, ObjAppSecure et VOPLogin qui sont toutes les trois masquées à l'utilisation et dans le

fichier d'archive vop.jar. Ni le client ni aucune implémentation n'auront réellement besoin de s'en soucier si on exclut la très légère gestion des VPOException.

Le seul fichier de configuration à considérer est vop.policy (section 5.4.2).

5.7 Utilisation des aspects

Si on prend un peu de recul face à l'implémentation et qu'on tente d'y retrouver des tendances de développement, des patrons en fait, on découvre quelques points très intéressants. Prenons le premier cas, celui de la classe InfoEtudiantAppSecure (section 5.2.3). Cette classe n'a qu'un rôle simple avec un nombre de possibilités limitées. Voyons deux exemples de méthode.

La méthode setNom(...) avec paramètres, sans retour

```
public void setNom(String nom) throws VOPException{
        Class<?> params = String.class;
        Object[] valeurs = {nom};
        execute("setNom", params, valeurs);
}
```

La méthode getNom(...) sans paramètres, avec retour

```
public String getNom() throws VOPException{
        return (String)execute("getNom");
}
```

On peut dégager la constatation suivante : toutes les méthodes contenues dans InfoEtudiantAppSecure peuvent être déduites en comparant l'interface de InfoEtudiantApp (l'ensemble de toutes les méthodes publiques de l'objet d'application du package vopimpl.infoetudiant) avec l'ensemble des méthodes implémentées rattachées à cet objet (package vopimpl.infoetudiant.vue). Chaque signature de méthode peut être synthétisée sous ce format

Analyse d'une signature de méthode de InfoEtudiantAppSecure

```
public <TYPE DE RETOUR> <NOM DE LA FONCTION>(<TYPE DU PARAM 1> <NOM DE LA
VARIABLE 1>,…,<TYPE DU PARAM N> <NOM DE LA VARIABLE N>)) throws VOPException{

        params = <TABLEAU DES TYPES DES N PARAMÈTRES DE LA SIGNATURE>
        valeurs = <TABLEAU DES N VARIABLES REÇU EN PARAMÈTRE>
        <TYPE DE RETOUR> execute(<NOM DE LA FONCTION>, params, valeurs);
}
```

Certaines manipulations de transformation seraient nécessaires pour gérer le passage de primitive à objet et vice-versa dans le cas des paramètres d'entrées et pour la valeur de retour. Java est un langage fortement typé ce qui, à notre avantage, force la définition des types à la compilation. Le nombre de types de primitive étant très limité et tout le reste étant des objets, nous pouvons conclure qu'il s'agit d'un exercice trivial pour tout bon programmeur.

Ayant établi un modèle de méthodes, rien ne nous empêche de prévoir une prochaine phase qui serait peut-être réalisée avec la programmation par aspect [Kiczales et al., 97] selon l'idée suivante.

Supposons l'appel à la méthode execute(...) comme une constante dans toutes les classes, elle peut être ciblée comme un point de coupe. On insère donc, avant le point de coupe, la gestion des paramètres d'entrée et, après le point, la gestion du retour d'appel.

Signature de la méthode

// Gestion des types des paramètres

// Gestion des paramètres

méthode execute(…) // Point de coupe

// Gestion du retour

Donc, il est réaliste de croire que la classe de liaison avec le client ne soit plus à coder éliminant ainsi toute possibilité d'erreur et améliorant la transparence du système. La classe « InfoEtudiantApp » est elle aussi intéressante. Reprenons le code de la méthode estOK(...) tel que présenté à la section 5.5.4.

On pourrait diviser le code en un bloc de sécurité, la vérification des accès aux vues et une implémentation logique du bloc fonctionnel V1 ET V2 ET V3. Il faudrait voir à remplacer le premier bloc de sécurité écrit manuellement et à le générer via la programmation par aspect à l'aide d'un point de coupe au début de la méthode. L'implémentation logique pourrait aussi être générée d'un moyen semblable ce qui nous laisserait un objet beaucoup plus dynamique. Le développement réel et la complexité en seraient encore diminués. Il s'agit de la méthode de faire traditionnelle de la programmation par vue.

5.8 La solution complète

Ayant abordé le côté logique de la solution en début de chapitre, voici deux diagrammes UML (figures 5.12 et 5.13) qui permettront de mieux visualiser l'organisation de la solution telle que développée. On peut aussi retrouver le code Java à l'annexe B.

Cette solution sera utilisée au chapitre 6 pour confronter les principes de sécurité appliqués à la programmation par vues face aux problématiques précédemment énoncées au chapitre 4.

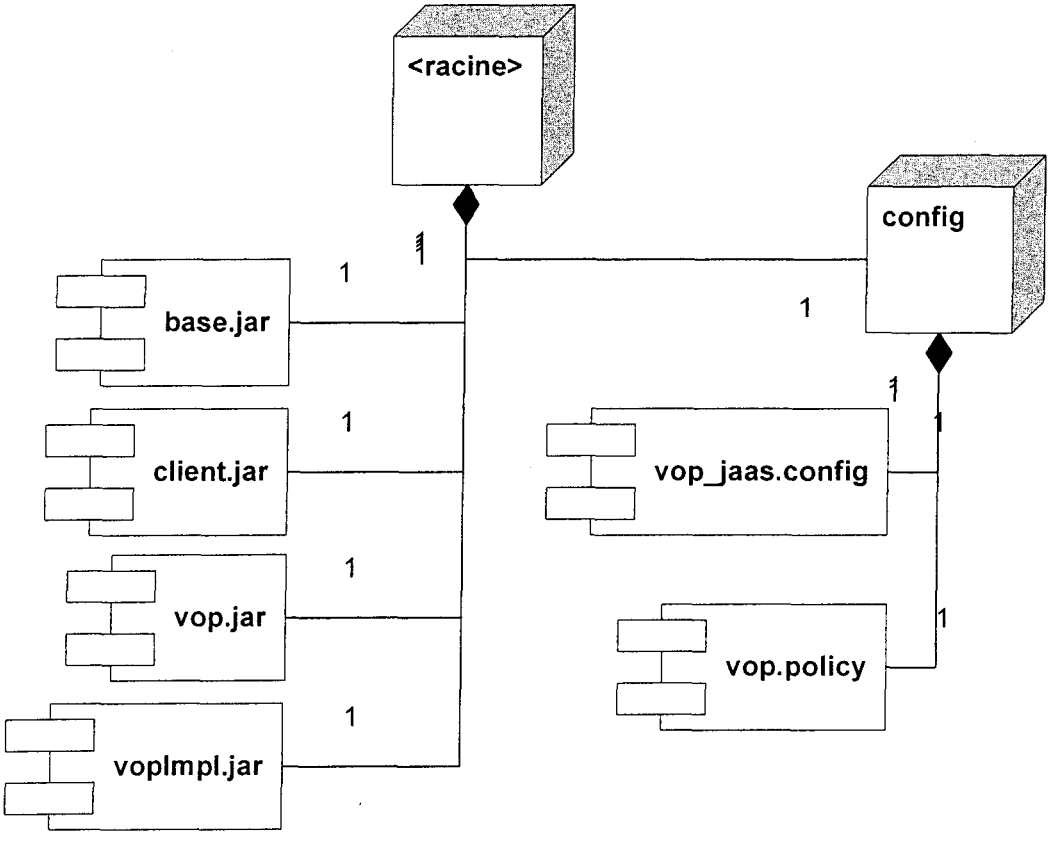

Figure 5.12 : Configuration de la solution complète

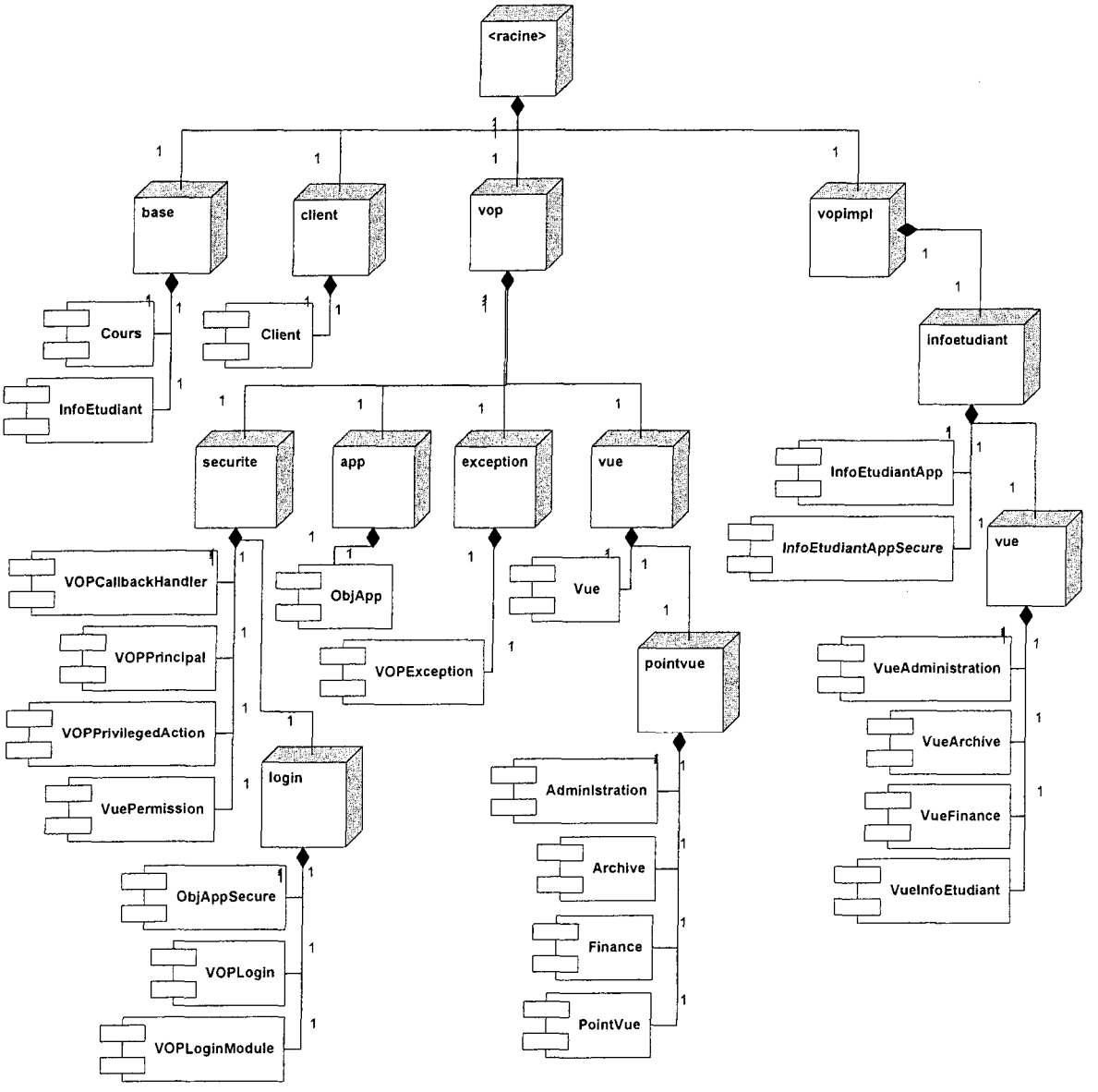

Figure 5.13 : Organisation des fichiers de la solution complète

6

RÉSULTATS ET ANALYSE

La solution proposée au chapitre précédent tentait de répondre à la problématique énoncée au chapitre 1. Avant de débuter l'analyse de la solution, il serait judicieux d'en revoir les grands principes.

L'objectif global étant de proposer une extension de sécurité au modèle de programmation par vue, trois objectifs spécifiques avaient été avancés :

- développer un modèle conceptuel de l'auto-adaptation des objets
- implémenter ce modèle pour faire évoluer les fonctionnalités des objets
- traiter un sous problème de sécurité. Il s'agit de trouver et implémenter un modèle de sécurité simple et léger pour préserver les droits d'accès de chaque programme client.

La solution a été développée avec succès en conservant l'essence de l'une approche sans modèle de sécurité. Nous pourrons aussi constater dans les prochaines sections que les principes fondamentaux de la programmation par vue ont été conservés et respectés (sections 2.3 et 4.1)

Dynamisme : Un objet peut changer de comportements en tout temps

Transparence : Le client est isolé des détails techniques de l'implémentation

Possibilité : La combinaison des vues permet une multitude d'utilisations fonctionnelles

Permanence : L'objet d'application et les vues conservent leur état entre les appels

Structure : Les appels de méthodes sont standardisés, solides et efficaces

Abstraction : Utilisation des points de vue

L'analyse de la solution répondra aux problématiques découlant de l'intégration du modèle de sécurité dans la programmation par vue (section 4.1) en respect des précédents objectifs, principes et contraintes.

6.1 La proposition à la problématique de l'approche actuelle

6.1.1 Le rôle du client

« Le rôle du client n'est pas adapté à une approche multi clients » (section 4.2.1)

Cette problématique fait ressortir le fait que chaque client doit être en parfait contrôle de l'état de ObjApp pour en assurer un fonctionnement adéquat. Avec une approche multiclients, ObjApp devient une ressource partagée dont l'état ne lui appartient pas. ObjApp ne peut garantir un service fiable à C1 en raison des activités de C2 par exemple.

Notre solution redonne à ObjApp le contrôle de son état. C1 et C2 ne peuvent plus gérer les vues de ObjApp. *Une vue est active, en même temps et ce, pour tous les clients possédant les permissions nécessaires.*

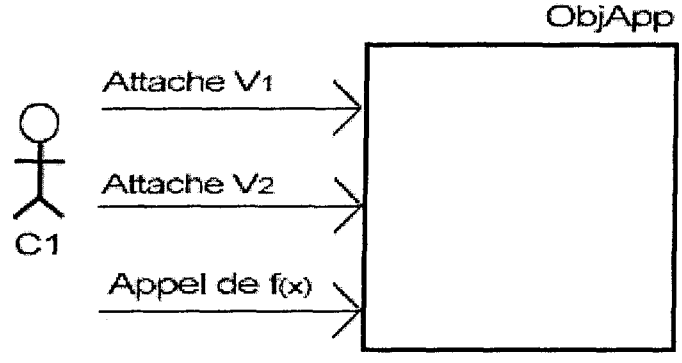

Figure 6.1 : Appel d'une méthode dans la programmation par vue

Dans la solution de départ (à la figure 6.1), C1 devait préparer lui-même l'état de l'objet d'application en attachant les vues nécessaires au bon fonctionnement de l'appel (au risque que V2 fasse de même!). Une fois ObjApp prêt à l'utilisation, C1 procédait à l'utilisation de l'objet à l'aide d'une série d'appels de méthode. Donc, pour chaque besoin fonctionnel particulier, le client devait connaître la combinaison de vues correspondante.

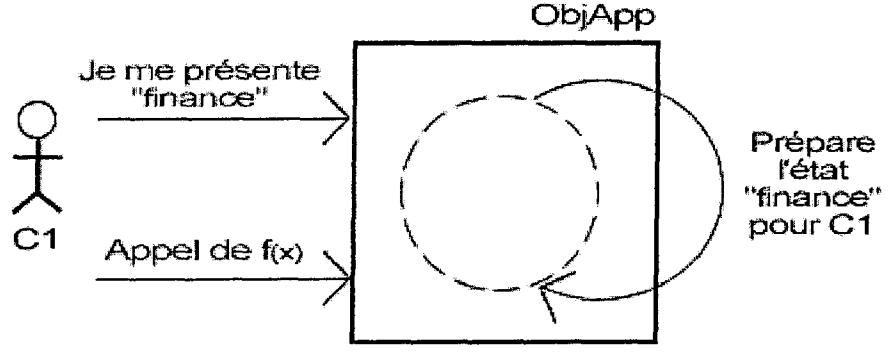

Figure 6.2 : Appel d'une méthode dans la programmation par vue avec modèle de sécurité

Pour éliminer cette complexité et améliorer le procédé, on demande plutôt au client de se présenter à ObjApp. Suivons l'exemple de la figure 6.2, C1 veut produire un rapport

financier et voudrait utiliser ObjBase en tant qu'entité financière. Il authentifie simplement son besoin, par exemple « finance », à l'objet d'application. À partir de ce moment, lorsque C1 utilisera ObjApp, ce dernier utilisera exclusivement l'interface regroupant les fonctionnalités des vues définies dans le profil fonctionnel « finance ». C'est une utilisation centralisée et transparente.

Simplement présentée, la solution peut paraître banale mais l'intégration de JAAS a été nécessaire pour y répondre. Un client peut maintenant être codé avec un besoin fonctionnel fixe (ex : finance) et l'objet d'application peut évoluer avec le temps avec un minimum de modification au code. On peut se souvenir que les vues qui seront associées à chaque profil lors de l'exécution sont gérées par le gestionnaire de sécurité, via les permissions sur les vues (VuePermission, section 5.4.1) et sont complètement décrites en dehors du code dans le fichier de configuration vop.policy (section 5.4.2)

6.1.2 Persistance des vues et permanence du profil

« La persistance de la vue est représentative des actions d'un seul client. Le partage des informations est source de confusion. De plus, le modèle actuel n'assure pas la permanence du profil usager » (voir sections 4.2.2 et 4.2.3)

ObjApp étant partagé par tous les clients, il est unique. Tant et aussi longtemps que nos vues implémentent strictement l'interface des points de vue, elles ne possèdent pas d'attributs, elles demeurent sans état. Toutefois, la situation illustrée par la figure 4.2 peut survenir.

Notre solution n'a pas répondu à ce cas particulier. Le problème reste entier et devra être géré par une implantation plus prudente. Pour en arriver à dupliquer un attribut, ObjApp devrait connaître et pouvoir reconnaître tous les clients qui l'utilisent, ce qui n'est malheureusement pas le cas. D'un appel à un autre, on ne garde pas de trace de l'appelant ce qui implique qu'il n'est pas possible de préserver un état réutilisable.

Une ébauche de solution avec profil a été considérée au début de l'exploration mais elle a été rapidement rejetée, car elle violait la contrainte de simplicité de l'ObjApp, il n'a pas le rôle de serveur mais bien d'objet fonctionnel. Deux solutions étaient possibles (voir figure 6.3). Soit on chargeait une instance de chaque vue pour chaque client ayant cette vue dans son profil, soit chaque attribut de chaque vue était dupliqué pour chaque client utilisant ObjApp.

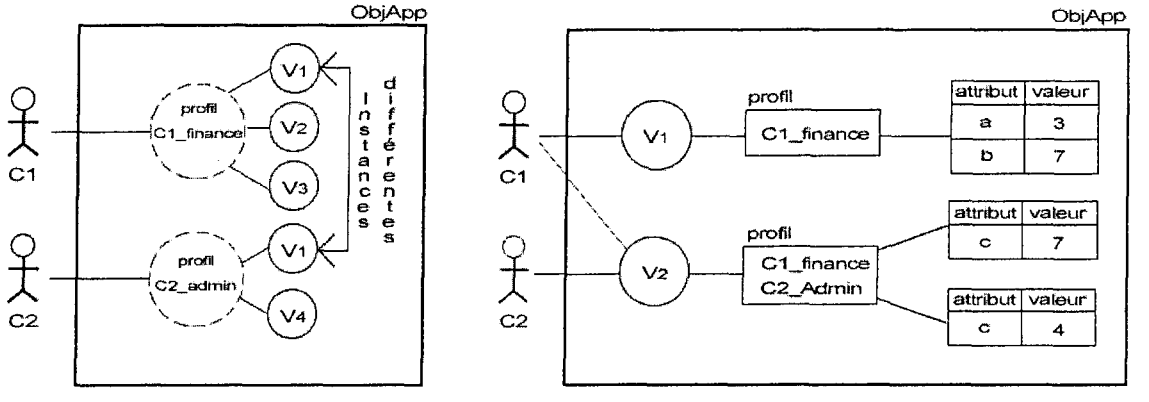

Figure 6.3 : Deux propositions de solution pour un profil dans ObjApp

ObjApp devrait alors connaître réellement C1 (une autre identité?) en plus de l'entité (Subject) fournie par JAAS. Dans l'implémentation actuelle, JAAS aurait été insuffisant pour répondre à cette demande. Il aurait fallu ajouter une authentification plus complexe pour le client et plus lourde à gérer pour ObjApp. Si cette problématique avait été

incontournable, l'utilisation d'un modèle de relation <client, <serveur, ObjApp>> tel que les EJB (Enterprise Java Bean) auraient pu être considéré.

6.1.3 Transparence d'utilisation des vues

« La dualité attachement/détachement et activation/désactivation du processus de gestion des vues représente une réalité informatique technique qui ne devrait pas être visible du côté du client » (section 4.2.4).

Dans l'approche traditionnelle de la programmation par vue, il était nécessaire d'attacher et ensuite d'activer une vue pour l'utiliser. Le client avait alors la possibilité de gérer l'activation de ses vues en fonction de ses besoins. Avec notre solution, cette option s'est avérée désuète : le contrôle des vues étant complètement transféré à ObjApp. *Les vues sont totalement transparentes du côté client. À la limite, ce dernier n'a plus conscience de leur existence.* L'utilisation de ObjAppSecure (section 5.2.3) isole complètement le client.

6.1.4 Évolution dynamique

« L'évolution des besoins est complètement dans les mains du client et le système de vue est tributaire et sans réel contrôle » (section 4.2.5).

Notre analyse de la solution nous a demandé de déterminer qui seraient en contrôle des vues. La solution actuelle laisse le contrôle au client mais, avec un objet d'application partagé, il est très difficile de synchroniser son utilisation. Comme précisé à la section précédente, *nous avons choisi de laisser la gestion du dynamisme des vues à ObjApp transformant du même coup le client en simple utilisateur.*

Il est possible qu'à des moments t_0 et t_1, ObjApp ne réagisse pas de la même façon à un appel de méthode particulier. C'est tout à fait logique si on continue de percevoir ObjApp comme un objet répondant aux besoins fonctionnels instantanés. Il possède en fait un état qui se détermine par l'organisation et la gestion des vues actives et accessibles à un moment donné. *Au fur et à mesure que ObjApp évolue fonctionnellement, son état s'ajuste.*

Cette problématique demande une gestion plus prudente de la part du client car il ne pourra pas déterminer à la compilation le comportement exact de ObjApp. De plus, ObjApp est aussi une simple classe, compilée et statique. Il est vrai que les permissions sont gérées via un fichier de configuration externe mais le besoin fonctionnel lui-même reste figé dans le code. Notre solution avance toutefois une alternative (section 5.6.2) mais elle n'a pas été développée.

6.1.5 Problèmes de droits

« Tous les clients peuvent gérer toutes les vues de l'objet d'application, sans restriction ni contrôle! » (section 4.2.6)

Sans extension de sécurité, il n'y a aucun moyen de réaliser l'authentification et l'autorisation donc aucune façon de différentier chacun des clients qui cherchent à accéder aux ressources. Avec les VuePermission (section 5.4.1), il est dorénavant impossible d'atteindre une ressource sécurisée sans l'autorisation préalable de ObjApp. Cette problématique à été au cœur du développement autour de JAAS et est complètement résolue.

On pourrait spécifier qu'il existe deux possibilités d'atteindre l'objet de base sans posséder de profil particulier. Tout d'abord, en atteignant l'objet de base directement, sans intermédiaire. Dans un second temps, en utilisant une méthode de ObjApp qui ne fait pas appel aux vues, une communication directe avec ObjApp. On en retrouve un exemple à la figure 2.2 en début de document.

6.1.6 Gestion des appels

« Comment écrire le code de f(x) dans ObjApp en tenant compte du nombre élevé de possibilités découlant des combinaisons (Cx, Vx) (où Cx est un client quelconque et Vx une vue quelconque) » (section 4.2.7)

On continue avec la problématique précédemment discutée à la section 6.1.4. Soyons concret. Si une fonction f(x) se retrouve dans 10 vues distinctes, il y aurait plusieurs dizaines de possibilités distinctes. En voici quelques exemples :

Voici le besoin fonctionnel à t_0

f(x) = V1.f(x) ET V2.f(x)

Voici le besoin fonctionnel à t_1

f(x) = V1.f(x) OU V3.f(x)

Voici le besoin fonctionnel à t_2

f(x) = (V1.f(x) + V2.f(x)) / V3.f(x) ET V4.f(x)

Voici le besoin fonctionnel à t_n

f(x) = V1.f(x) ET V2.f(x) ET V3.f(x) ET V4.f(x) ET V5.f(x) ET V6.f(x) ET V7.f(x) ET V8.f(x) ET V9.f(x) ET V10.f(x)

Sans oublier la gestion des permissions pour chaque vue. Est-ce une vue optionnelle ou obligatoire? Un exemple concret se retrouve à la section 5.5.4.

La vraie question est de savoir si on doit recompiler ObjApp à chaque modification du besoin ou encore coder toutes les possibilités pour ensuite aiguiller les demandes? Il serait envisageable de prévoir un mécanisme externe tel que celui schématisé à la figure 6.4. Le traitement des vues pourrait être effectué par un module externe (un script en Python pour définir des règles d'affaire?)

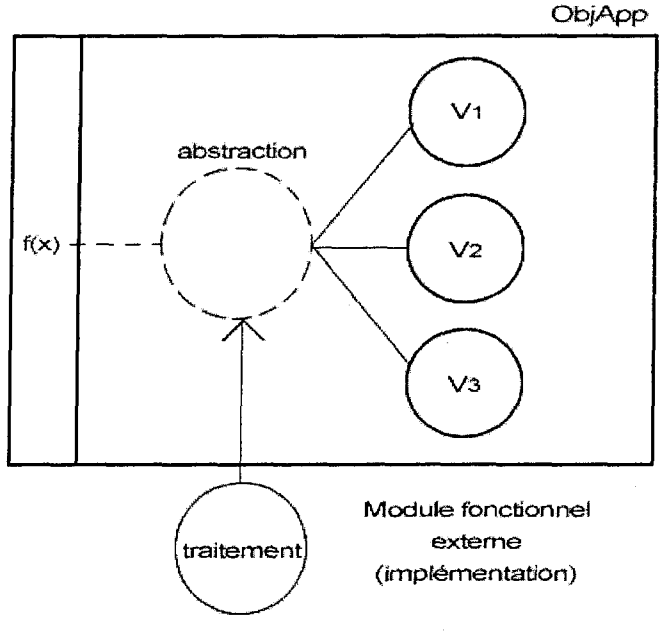

Figure 6.4 : Proposition de solution à la problématique de gestion des appels

Toutefois, il n'est pas évident que cet effort de développement aide à la simplification de l'utilisation. Notre solution conserve malgré tout son modèle de gestion statique qui démontre le bien-fondé du modèle de sécurité dans la programmation par vue.

6.2 La proposition à la problématique de délégation

La problématique énoncée à la section 4.3 propose trois solutions. *Par souci de standardisation et de sécurité, notre solution utilisera donc une délégation hybride.* Nous utiliserons respectivement les délégations brisée et partielle selon la règle suivante (figure 6.5):

Si la méthode appelée est dans la vue

Utiliser la délégation brisée

Autrement

Utiliser la délégation partielle

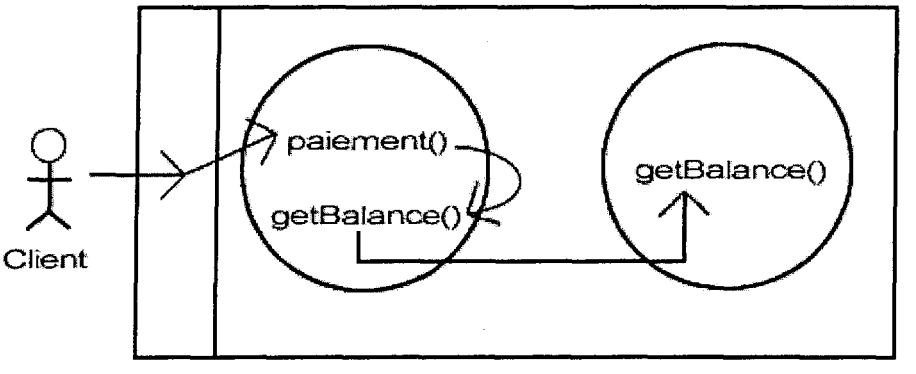

Figure 6.5 : La délégation hybride

Voici un exemple de code en complément à ceux présentés à la section 4.3

La méthode paiement(...) avec délégation hybride

```
public void paiement (float montant){
        objBase.setBalance(getBalance() - montant);
}
```

On retourne toujours à la vue en premier lieu en cas d'appel de méthode à l'intérieur de la vue elle-même car c'est à cet endroit qu'est fixée l'instance de VuePermission (voir figure 4.8)

Les règles fonctionnelles implantées par les vues pourront ainsi être utilisées à leur maximum. De plus, rien n'empêche l'utilisation d'une autre approche dans le développement des vues, il s'agit plutôt d'une bonne pratique de programmation.

6.3 La proposition à la problématique de sécurité

JAAS a été la solution à cette problématique. D'autres techniques, telles que les EJB, auraient pu être utilisées avec succès mais nous voulions rester simples et éviter de surcharger le client de procédure d'utilisation et d'installation. De plus, la sécurité au sens large n'est pas le réel besoin d'ObjApp, il est plutôt question d'authentifier l'état d'un client (ex : finance, administrateur, anonyme) et de l'autoriser à ses vues pour définir une réponse répondant à un besoin. En résumé, la sécurité avec la programmation par vue est assurée par cette liste d'énoncés :

- Le client est identifié (voir VOPLogin à la section 5.2.1)
- Le client est identifiable (voir Subject/VOPPrincipal à la section 5.2.2)
- La signature des méthodes reste la même pour le client (voir ObjAppSecure à la section 5.2.3)
- La communication entre le client et l'objet d'application est efficace et transparente (voir ObjApp et la méthode exécute(...) à la section 5.5.2)

- L'objet d'application reconnaît le client et a la possibilité de gérer les actions (voir VOPPrivilegedAction à la section 5.5.3)
- Les vues sont protégées (voir VuePermission à la section 5.4.1)
- L'exécution d'un appel sécurisé est dynamique selon le besoin (voir gestionnaire de sécurité et fichier de politique de sécurité aux sections 3.5 et 5.4.2)
- Le retour à l'usager est clair et efficace (voir VOPException à la section 5.5.2)

L'analyse du modèle de sécurité s'est effectuée au cours de sa présentation au chapitre 4. Après plusieurs itérations, séries de test, vérifications et questionnements face aux contraintes émises en début de chapitre, nous en sommes venus à la conclusion que son intégration à la programmation par vue a réussi. La figure 6.6 fait office de témoignage en illustrant l'ampleur du modèle de sécurité vis-à-vis la solution complète, le cercle en pointillé représente la zone de sécurité, invisible au client et à l'objet de base.

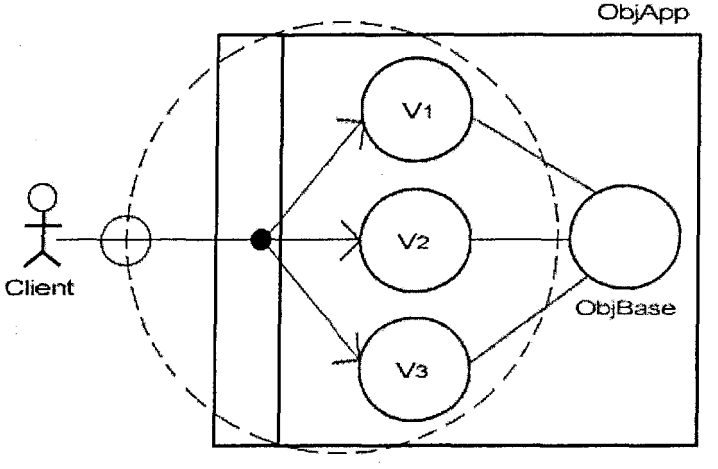

Figure 6.6 : Ampleur du modèle de sécurité

6.4 La proposition à la problématique de synchronisation

Reprenons les propos de la section 4.5. La synchronisation est une problématique très épineuse de tout système pouvant être utilisé simultanément par plusieurs acteurs. Dans un scénario de test avec une faible utilisation (notre cas), il est improbable que le problème soit constaté.

Dans la réalité, un objet peut être victime de son succès et subir une forme demande. Notre simple exmple de solution quiu se retrouuve à l'annexe B, n'est pas synchronisée mais nous avons tout de même des avenues de solutions. Voici quelques pistes possibles imaginées, mais non testées, lors du développement.

Solution #1 : Créer un nouvel objet ObjBaseSync qui englobe complètement ObjBase et synchronise tous les accès... Chaque ObjBase aurait-il son propre ObjAppSync ou il s'agirait d'une classe générique (voir figure 6.7)?

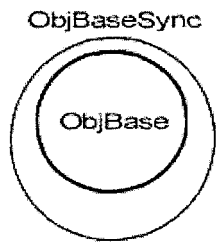

Figure 6.7 : ObjAppSync et ObjApp

Solution #2 : Synchroniser la méthode execute(...) dans ObjApp. Cette solution fonctionne bien mais elle a un désavantage majeur : tous les appels de tous les clients seraient traités en série plutôt qu'en parallèle. Avec un volume d'appels à ObjApp suffisant, le système

sera considérablement congestionné (figure 6.8). Malgré cet inconvénient, cette solution est celle retenue.

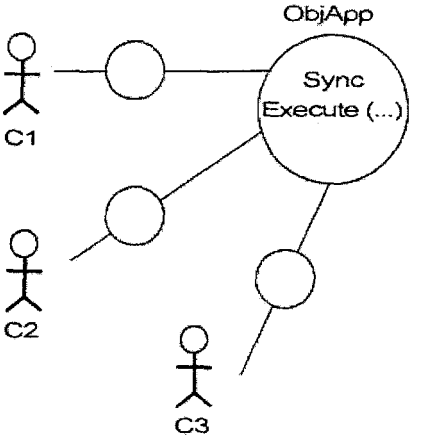

Figure 6.8 : Synchronisation de la méthode execute(...)

Solution #3 : La troisième approche consiste à synchroniser chaque méthode dans ObjAppImpl. On relègue l'implémentation au développeur de chaque nouvel objet d'application. Est-ce un risque? Elle apporte aussi quelques risques de barrure (*lock*) avec les vues. Dans l'exemple de la figure 6.9, si C1 appelle une méthode f(x) synchronisée et C2 appelle simultanément une méthode g(x) synchronisée, on risque d'avoir de sérieux problèmes...

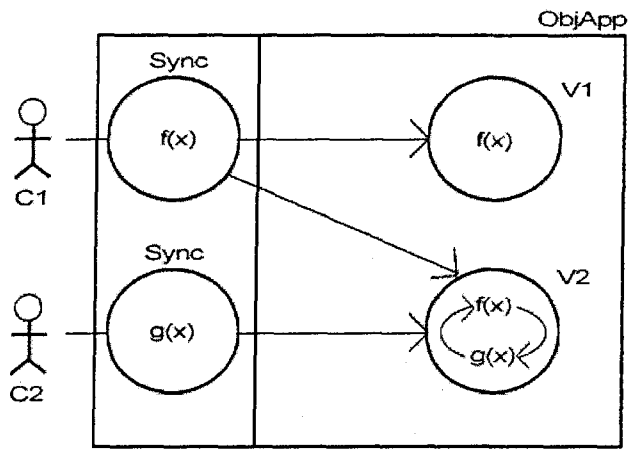

Figure 6.9 : Synchronisation unitaire dans ObjAppImpl

6.5 La proposition à la problématique d'exécution partielle

Notre solution ne gère pas les conflits d'exécution partielle, car *cette problématique relève de l'implémentation* et contribue à complexifier le code. Reprenons l'exemple débuté de la section 4.6.

ObjApp.f(x)

Retourne V1.f(x) ET V2.f(x) ET V3.f(x)

Considérons que chaque appel de f(x) modifie l'état de ObjBase. Si C1 n'a pas accès à V3, il faut éviter l'exécution de V1.f(x) et V2.f(x) car il n'est pas possible de revenir en arrière. En guide de solution, voici donc un exemple de code simplifié de ce que pourrait être la solution.

Exemple de solution au problème d'exécution partielle

```
public bool f(int x) throws VOPException {

        verifierVue("V1");
        verifierVue("V2");
        verifierVue("V3");
        return getV1().f(x) && getV2.f(x) && getV3.f(x);
    }
```

On ajoute une méthode verifierVue(...) dans ObjApp qui valide les accès à la vue. Le code de f(x) valide que le client à bien accès aux vues obligatoires dès le début de la méthode, avant de procéder aux appels. Elle ne retourne rien à part une VOPException en cas de manque d'autorisation.

7

CONCLUSION

Les problématiques soulevées dans ce document peuvent être regroupées en une problématique plus globale qui touche beaucoup de système : comment plusieurs clients peuvent-ils utiliser sans crainte un système en évolution? Si les besoins des clients changent et que le système évolue, comment garder un lien de communication sans faille? Notre solution propose donc, à sa manière, une réponse à cette épineuse problématique. Revoyons donc les principales embuches et réussites rencontrées au cours du développement de notre outil.

Tout d'abord, l'ajout d'une extension de sécurité à la programmation par vue s'est avéré plus difficile que prévu. La raison se retrouve à travers les discussions concernant les sous problématiques décrites et analysées dans les deux derniers chapitres. Toute la question de l'intégration de l'authentification et l'autorisation avec JAAS était tout de même assez bien documentée sur les sites de SUN mais on était encore loin d'une adaptation à la solution.

Le concept d'accès aux vues n'a pas été de tout repos à intégrer au système. JAAS permet de valider l'accès aux ressources du système, à des objets et dans notre cas à des vues. Toutefois, notre besoin dépasse la simple gestion des vues. Notre client ne va pas seulement accéder à une ou plusieurs vues, il appelle une méthode f(x) de l'objet

d'application qui répondra à son besoin. La jonction entre les vues et leurs méthodes a été réussie grâce au *reflexion package* de Java, implémentation que l'on peut retrouver dans les méthodes execute(...) des classes ObjAppSecure et ObjApp.

Avec, au départ, un minimum d'expérience avec JAAS et avec le *reflexion package*, les premiers pas ont été faits à travers un réel champ de mines.

Une fois la solution JAAS sélectionnée et testée, il demeurait encore l'épreuve reposant sur le questionnement de ObjApp par le client. Un client appelait f(x) et devait conserver le même appel avec l'ajout de sécurité. L'intégration de ObjAppSecure avec son rôle d'encapsulation a permis d'isoler le client. Des tests non concluants avec des clés RSA et du client-serveur avec le protocole RMI ont été abandonnés.

Notre contribution à l'évolution des techniques de gestion des comportements (voir section 1.3) permet d'offrir une vision différente de l'utilisation de la programmation par vue. Résumons-en donc les principaux points :

- Nous avons réussi à intégrer un outil de gestion dynamique des vues qui respecte les principales contraintes de simplicité et de transparence de la programmation par vues.
- L'analyse objet a été réalisée à travers plusieurs séries de raffinements afin de produire une solution, quoique complexe en son cœur, reste facile à implémenter en Java mais qui serait aussi portable dans d'autres langages objets. Nous proposons un outil complet, analysé, développé et testé.
- La déresponsabilisation du client face à l'évolution des comportements du système permet une utilisation transparente de l'objet de base. Notre proposition utilise les

techniques d'identification et d'authentification de JAAS, ce qui introduit un modèle de permission plus simple et raisonnable.

- Notre solution résoud également une importante problématique reliée à la dissociation programmaticale des comportements de l'objet d'application face aux préoccupations d'accès aux méthodes. Un objet peut maintenant évoluer dans ses comportements sans être recodé à chaque fois.

Trois articles ont déjà été publiés sur le sujet [Mcheick et Dallaire, 08], [Mcheick et al., 08] [Mcheick et al., 09].

Au début du chapitre 6, nous rappelions les options et contraintes du projet, il est maintenant grand temps d'affirmer que la solution a atteint les objectifs fixés. Toutefois, il reste toujours des zones grises qui pourraient être explorées à travers des recherches futures. Voici quelques unes de nos suggestions

- Il serait intéressant de pousser l'intégration de la programmation par aspects. La sécurité dans ObjAppImpl demande assez peu de code et les points de coupe peuvent être aisément identifiés. La gestion de la sécurité à l'intérieur de ObjAppImpl pourrait être dissociée du besoin. En ce moment, les validations sont mélangées au code fonctionnel ce qui n'a pas de raison d'être.
- La classe ObjAppSecureImpl pourrait aussi être générée automatiquement telle que suggérée à la section 5.7.
- Afin de valider la robustesse de l'approche, la solution pourrait être adaptée à un cas de jeux vidéo où les accès seraient plus rapides et plus fréquents. Des

approfondissements au niveau de la synchonisation et de la distribution de l'information seraient nécessaires.

- Serait-il possible d'associer l'évolution des comportements à une intelligence artificielle?

Pour conclure et pour divertir le lecteur, nous avons ajouté en annexe un exemple ludique de programmation par vue dans l'univers des jeux vidéo : L'Oracle du temps!

Bibliographie

[Chan, 02] Chan, P., The Java Developers Almanac, volume 2, Addison Wesley ,2002, 1024 p.

[Constantinides et Skotiniotis, 2004] Constantinides C., Skotiniotis T., Providing Multidimensional Decomposition In Object-Oriented Analysis and Design, *Preceddings of the IASTED International Conference*, pp.101-110, Innsbruck, Austria, Février 17-19, 2004.

[CSS, 01] College of Computer Science, Subject Oriented Programming (SOP) http://www.ccs.neu.edu/research/demeter/SOP/

[Dargham, 01] Joumana D., Programmation par vues : Fondements et implementation des vues, Thèse de doctorat, Université de Montréal, 2006, 162p.

[DotNetGuru, 03] AOP, Intérêt et usage http://www.dotnetguru.org/articles/dossiers/aop/quid/AOP15.htm

[Eclipse, 03] The AspectJ Programming Guide http://www.eclipse.org/aspectj/doc/released/progguide/index.html

[Fink et al., 03] Fink T., Koch M., Oancea C., Specification and Enforcement of Access Control in Heterogeneous Distributed Applications, ICWS-Europe 2003, LNCS 2853, pp. 88–100, 2003

[Gardarin, 01] Gardarin, G., Bases de données, objets et relationnel, Eyrolles, 2001, 788p.

[Garms, 01] Garms J., Somerfield D., Professional Java Security, Wrox Press, mai 2001, 521p.

[Harrison et Ossher, 1993] Harrisin W., Ossher H., Subject-oriented programming: a critique of pure objects, in Proceedings of OOPSLA'93, pp.411-428, Washington D.C, USA, Septembre 26-Octobre 1, 1993

[Holford et al., 04] Holford J. W., Caelli W. J., Rhodes A., Using self-defending objects to develop security aware applications in Java, 27th Australasian Computer Science Conference, 2004, 9p.

[IETF, 97] RFC 2251 : Lightweight Directory Access Protocol (v3) http://www.ietf.org/rfc/rfc2251.txt

[Kiczales et al., 97] Kiczales G., Lamping J., Mendekar A., Maeda C., Videira Lopes C., Loingtier J.M., Irvin J., Aspect-Oriented Programming, in proceedings of the European Conference on Object-Oriented Programming (ECOOP, Finland, Springer-Verlag LCNS 1241m p.220-242, juin 1997.

[Mcheick, 06] Mcheick H., Distribution d'objets en utilisant les techniques de développement orientées aspect : programmation orientée aspect, programmation orientée sujet et programmation orientée vue. Thèse de doctorat, Université de Montréal, 2006, 273 p.

[Mcheick et Dallaire, 08] Mcheick H., Dallaire E., SecurityViews: Security Object Model for Views-Oriented Programming, MCTECH, 2008, 5p.

[Mcheick et al., 08] Mcheick H., Mili H., Dallaire E., View-based privacy model for Object

	Systems, IADIS, 2008, 10p.
[Mcheick et al., 09]	Mcheick H., Mili H., Dallaire E., First Step of Security Model for Separation of Concerns, AICCSA, 2009, 7p.
[Microsoft, 09]	Microsoft Technet, Understanding Role Based Access Control, http://technet.microsoft.com/en-us/library/dd298183.aspx
[Mili et al., 01]	Mili H., MCheick H., Dargham J., Delloul S., Distribution d'objets avec vues, in Procedings of LMO'01 (Langage et modèle objet) vol. 7 no. 1, 2/2001 eds Hermès, pp.27.44 le Croisic France, janvier 2001.
[Mili et al., 99]	Hafedh M., Mili A., Dargham J., Cherkaoui O., Godin R., View Programming : Towards a framework for Decentralized Development And Execution of OO Programs, proceedings of TOOLS USA'99, Prentice-Hall, p.211-221, 1999.
[Mili et al., 06]	Mili H., Sahraoui H., Lounis H., Mcheick H., Elkharraz A., Concerned about separation, Fundamental approches to software engineering (FASE2006), Vienna, Autriche, 2006.
[NIST, 09]	National Institute os Standards and Technology, Role Based Access Control (RBAC) and Role Based Security http://csrc.nist.gov/groups/SNS/rbac/
[Oaks, 01]	Oaks, S., Java Security, 2nd Edition, O'Reilly, 2001, 482p.
[Plouin et al., 04]	Plouin G. Soyer, J., Trioullier M.-E., Sécurité des architectures WEB, Dunod, 2004, 486p.
[Sebesta, 05]	Sebesta, R., Concepts of Programming Languages, 7th edition, Addison Wesley, 2005, 724 p.
[Stroustrup, 07-1]	Bjarne Stroustrup's homepage http://www.research.att.com/~bs/homepage.html
[Stroustrup, 07-2]	What is so great about classes? http://www.research.att.com/~bs/bs_faq.html#class
[SUN, 07]	Utilisation du modèle de sécurité de Java, http://java.sun.com/docs/books/tutorial/security/TOC.html, 2007
[SUN, 06]	Guide de sécurité de Java 1.6 http://java.sun.com/javase/6/docs/technotes/guides/security/, 2006
[SUN, 06-1]	Java Authentication and Authorization Service (JAAS) Reference Guide for the Java SE Development Kit 6 http://java.sun.com/javase/6/docs/technotes/guides/security/jaas /JAASRefGuide.html, 2006
[SUN, 06-2]	JAAS Authentication Tutorial http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/tutorials /GeneralAcnOnly.html, 2006
[SUN, 06-3]	JAAS Authorization Tutorial http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/tutorials /GeneralAcnAndAzn.html, 2006
[SUN, 06-4]	Java Platform, Standard Edition 6 API Specification http://java.sun.com/javase/6/docs/api/, 2006
[SUN, 06-5]	Java Platform, Standard Edition 6 VM Specification

http://java.sun.com/javase/6/docs/technotes/guides/vm/index.html, 2006

[SUN, 04] Common Development and Distribution License (CDDL) http://www.sun.com/cddl/cddl.html, 2004

ANNEXES

A. Le code « InfoEtudiant »

Cette section de l'annexe contient le code Java de l'exemple qui a été utilisé pour faire la présentation de la programmation par vue dans le chapitre d'introduction et au chapitre 2. Certaines parties du code ont été éliminées pour aider à la lecture. Cet exemple ne contient aucune notion reliée à la sécurité et correspond à une version allégée et adaptée de la solution de Hamid Mcheick [Mcheick, 06] qui a servi de solution de départ à l'implantation de notre solution. Voir le diagramme de classes de la figure 2.9 pour plus de détail.

InfoEtudiantApp

```
public class InfoEtudiantApp extends ObjApp {

    public InfoEtudiantApp(Object monObjBase){
        super(monObjBase);
    }
    private InfoEtudiant getObjBase(){
        return (InfoEtudiant)objBase;
    }
    private VueAdministration getVueAdministration(){
        return (VueAdministration)getVue("Administration");
    }
    private VueFinance getVueFinance(){
        return (VueFinance)getVue("Finance");
    }
    private VueArchive getVueArchive(){
        return (VueArchive)getVue("Archive");
    }
    public boolean estOK(){
        boolean ok = true;
        if (vueEstAttachee("Administration"))
            ok &= getVueAdministration().estOK();
        if (vueEstAttachee("Finance"))
            ok &= getVueFinance().estOK();
        if (vueEstAttachee("Archive"))
```

```
            ok &= getVueArchive().estOK();
        return ok;
    }
    public void setNom(String nom) throws VOPException{
        if (vueEstAttachee("Administration"))
            getVueAdministration().setNom(nom);
        else
            throw new VOPException("Aucune vue n'est active,
            impossible d'exécuter la méthode setNom()");
    }
    public Vector getComposantes() throws VOPException{
        if (vueEstAttachee("Administration"))
            return getVueAdministration().getComposantes();
        throw new VOPException("Aucune vue n'est active,
        impossible d'exécuter la méthode getComposantes()");
    }
    public boolean enDette() throws VOPException{
        if (vueEstAttachee("Administration"))
            return getVueAdministration().enDette();
        throw new VOPException("Aucune vue n'est active,
        impossible d'exécuter la méthode enDette()");
    }
    public void setStatut(String statut) throws VOPException{
        if (vueEstAttachee("Administration"))
            getVueAdministration().setStatut(statut);
        else
            throw new VOPException("Aucune vue n'est active,
            impossible d'exécuter la méthode setStatut()");
    }
    public void afficherComposantes() throws VOPException{
        if (vueEstAttachee("Archive"))
            getVueArchive().afficherComposantes();
        else
            throw new VOPException("Aucune vue n'est active,
            impossible d'exécuter la méthode afficherComposantes()");
    }
    public float getBalance() throws VOPException{
        float balance = getObjBase().getBalance();
        if (vueEstAttachee("Finance"))
            if (balance != getVueFinance().getBalance())
                throw new VOPException("Impossible de balancer avec la vue Finance");
        if (vueEstAttachee("Archive"))
            if (balance != getVueArchive().getBalance())
                throw new VOPException("Impossible de balancer avec la vue Archive");
        return balance;
    }
    public void paiement(float montant) throws VOPException{
        if (vueEstAttachee("Finance"))
            getVueFinance().paiement(montant);
        else
            throw new VOPException("Aucune vue n'est active,
            impossible d'exécuter la méthode paiement()");
    }
    public String getStatut(){

        return getObjBase().getStatut();
    }
}
```

ObjApp

```
public abstract class ObjApp {

    private Hashtable vues;
    protected Object objBase;

    public ObjApp(Object monObjBase){
        vues = new Hashtable();
        objBase = monObjBase;
    }

    public void attacherVue(String vue){
        try{
            Class nouvelleClasse = Class.forName("vue.Vue" + vue);
            Constructor[] constructors = nouvelleClasse.getConstructors();
            Vue nouvelleVue = (Vue)constructors[0].newInstance(objBase, this);
            vues.put(vue, nouvelleVue);
        }
        catch (Exception e){}
    }

    public void detacherVue(String vue){
        vues.remove(vue);
    }

    protected Vue getVue(String vue){
        return (Vue)vues.get(vue);
    }

    public boolean vueEstAttachee(String vue){
        return vues.containsKey(vue);
    }
}
```

InfoEtudiant

```
public class InfoEtudiant {

    private String nom;
    private float balance;
    private String statut;
    private Vector cours;

    public InfoEtudiant(String monNom, float maBalance, String monStatut){
        nom = monNom;
        balance = maBalance;
        statut = monStatut;
        cours = new Vector();
        initCours();
    }

    private void initCours(){
        cours.add(new Cours("Programmation orientée objet", "100", "A+"));
        cours.add(new Cours("Analyse Objet", "107", "B"));
        cours.add(new Cours("Projet", "113", ""));
    }

    public void setNom(String nouveauNom){
        nom = nouveauNom;
    }

    public void setBalance(float nouvelleBalance){
        balance = nouvelleBalance;
    }

    public void setStatut(String nouveauStatut){
        statut = nouveauStatut;
    }

    public String getNom(){
        return nom;
    }
```

```
    public float getBalance(){
        return balance;
    }
    public String getStatut(){
        return statut;
    }
    public Vector getListeCours(){
        return cours;
    }
}
```

Cours

```
public class Cours {

    private String nom;
    private String sigle;
    private String cote;

    public Cours(String nomCours, String sigleCours, String coteCours){
        nom = nomCours;
        sigle = sigleCours;
        cote = coteCours;
    }
    public String getNom(){
        return nom;
    }
    public String getSigle(){
        return sigle;
    }
    public String getCote(){
        return cote;
    }
    public void setCote(String nouvelleCote){
        cote = nouvelleCote;
    }
}
```

Client

```
public class Client {

    public static void main(String argv[]){
        InfoEtudiant infoEtudiant = new InfoEtudiant("Dallaire Eric", -100, "maitrise");
        InfoEtudiantApp objApp = new InfoEtudiantApp(infoEtudiant);

        try{
            System.out.println(objApp.estOK());
            System.out.println(objApp.getBalance());
            System.out.println(objApp.getStatut());
            //objApp.afficherComposantes();

            objApp.attacherVue("Administration");
            objApp.attacherVue("Finance");
            objApp.attacherVue("Archive");
            System.out.println(objApp.estOK());
            objApp.afficherComposantes();
        }
        catch(VOPException vope){
            System.out.println(vope.getMessage());
            vope.printStackTrace();
        }
    }
}
```

VOPException

```
public class VOPException extends Exception{
    public VOPException(String message){
        super(message);
    }
}
```

PointVue

```
public interface PointVue {
    // Aucune implémentation, il s'agit seulement d'une déclaration de principe
}
```

Finance

```
public interface Finance extends PointVue{
    public abstract float getBalance();
    public abstract boolean estOK();
    public abstract void paiement(float montant);
}
```

Archive

```
public interface Archive extends PointVue{
    public abstract void afficherComposantes();
    public abstract float getBalance();
    public abstract boolean estOK();
}
```

Administration

```
public interface Administration extends PointVue{
    public abstract boolean estOK();
    public abstract void setNom(String nom);
    public abstract Vector getComposantes();
    public abstract boolean enDette();
    public abstract void setStatut(String statut);
}
```

Vue

```
public abstract class Vue {
    protected Object objBase;
    protected ObjApp objApp;
    public Vue(Object monObjBase, ObjApp monObjApp){
        objBase = monObjBase;
        objApp = monObjApp;
    }
}
```

VueInfoEtudiant

```
public abstract class VueInfoEtudiant extends Vue {
    public VueInfoEtudiant (Object monObjBase, ObjApp monObjApp){
        super(monObjBase, monObjApp);
    }
    protected InfoEtudiant getObjBase(){
        return (InfoEtudiant)objBase;
    }
```

```
    protected InfoEtudiantApp getObjApp(){
        return (InfoEtudiantApp)objApp;
    }
}
```

VueAdministration

```
public class VueAdministration extends VueInfoEtudiant implements Administration{
    public VueAdministration (Object monObjBase, ObjApp monObjApp){
        super(monObjBase, monObjApp);
    }
    public boolean estOK(){
        return !getComposantes().isEmpty();
    }
    public void setNom(String nom){
        getObjBase().setNom(nom);
    }
    public Vector getComposantes(){
        return getObjBase().getListeCours();
    }
    public boolean enDette(){
        return getObjBase().getBalance() < 0;
    }
    public void setStatut(String statut){
        getObjBase().setStatut(statut);
    }
}
```

VueArchive

```
public class VueArchive extends VueInfoEtudiant implements Archive {
    public VueArchive (Object monObjBase, ObjApp monObjApp){
        super(monObjBase, monObjApp);
    }
    public void afficherComposantes(){
        Vector composantes = getObjBase().getListeCours();
        for (int x = 0; x < composantes.size(); x++){
            Cours cours = (Cours)composantes.get(x);
            System.out.println(cours.getNom());
            System.out.println(cours.getSigle());
            System.out.println(cours.getCote());
        }
    }
    public float getBalance(){
        return getObjBase().getBalance();
    }
    public boolean estOK(){
        return getObjBase().getStatut().compareTo("") != 0;
    }
}
```

VueArchive

```
public class VueFinance extends VueInfoEtudiant implements Finance {
    public VueFinance (Object monObjBase, ObjApp monObjApp){
        super(monObjBase, monObjApp);
    }
    public float getBalance(){
        return getObjBase().getBalance();
    }
```

```
    public boolean estOK(){
        return getBalance() == 0;
    }
    public void paiement(float montant){
        getObjBase().setBalance(getBalance() - montant);
    }
}
```

B. Le code « InfoEtudiant avec modèle de sécurité »

Cette section de l'annexe contient le code Java de l'exemple qui a été utilisé pour faire la présentation de la programmation par vue avec modèle de sécurité utilisé et présenté à travers les chapitres 3 et subséquent. La solution de départ est présentée à l'annexe A et on peut revoir les figures 5.1, 5.3 et 5.13 pour visualiser l'organisation des classes.

base.Cours

```
Identique à la classe de la solution de base
```

base.InfoEtudiant

```
Identique à la classe de la solution de base
```

client.Client

```
public class Client {
    public static void main(String[] args){
        System.out.println("\n_____\nAUTHENTIFICATION\n_____");
        InfoEtudiantAppSecure infoEtudiantAppSecure = null;
        try{
            infoEtudiantAppSecure = (InfoEtudiantAppSecure)VOPLogin.login("InfoEtudiant");
        }
        catch(VOPException vope){
            vope.printStackTrace();
            return;
        }
        System.out.println("\n_____\nTEST #1\n_____");
        try{
            boolean estOK = infoEtudiantAppSecure.estOK();
            System.out.println("* estOK   : " + estOK);
        }
        catch(VOPException vope){
            vope.printStackTrace();
        }
        try{
            System.out.println("\n_____\nTEST #2\n_____");
            try{
                System.out.println("* Le nom est : " + infoEtudiantAppSecure.getNom());
                System.out.print("* Entrer un nouveau nom   : ");
```

```
                infoEtudiantAppSecure.setNom((new BufferedReader
                (new InputStreamReader(System.in))).readLine());
                System.out.println("* Le nom est : " + infoEtudiantAppSecure.getNom());
            }
            catch(IOException ioe){
                ioe.printStackTrace();
            }
        }
        catch(VOPException vope){
            vope.printStackTrace();
        }
        try{
            System.out.println("\n_______\nTEST #3\n_______");
            System.out.println("* La balance est de : " +
            infoEtudiantAppSecure.getBalance());
        }
        catch(VOPException vope){
            vope.printStackTrace();
        }
        try{
            System.out.println("\n_______\nTEST #4\n_______");
            Vector composantes = infoEtudiantAppSecure.getComposantes();
            System.out.println("* La liste des composantes (" + composantes.size() + ")");
            for (int x = 0; x < composantes.size(); x++)
                System.out.println("  * Composante # " + x + " : " +
                ((Cours)composantes.get(x)).toString());
            ((Cours)composantes.get(2)).setCote("C-");
            System.out.println("\n  * Composante # 2 : " +
            ((Cours)composantes.get(2)).toString());
        }
        catch(VOPException vope){
            vope.printStackTrace();
        }
        try{
            System.out.println("\n_______\nTEST #5\n_______");
            infoEtudiantAppSecure.afficherComposantes();
        }
        catch(VOPException vope){
            vope.printStackTrace();
        }

        System.out.println("\n_______\nTEST #6\n_______");
        try{
            System.out.println("* enDette    : " + infoEtudiantAppSecure.enDette());
            infoEtudiantAppSecure.paiement((float)150.45);
            System.out.println("* enDette    : " + infoEtudiantAppSecure.enDette());

        }
        catch(VOPException vope){
            vope.printStackTrace();
        }
        System.out.println("\n_________________\nFIN DES TESTS\n_________________");
        try{
            System.in.read();
        } catch (IOException ex) {
            Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

vop.app.ObjApp

```
public abstract class ObjApp {

    private Hashtable vues;
    protected Object objBase

    public ObjApp(Object monObjBase){
        vues = new Hashtable();
        objBase = monObjBase;
    }

    private void attacherVue(String vue) throws VOPException{
        try{
            String nomClasse = objBase.getClass().getName();
            int debutNom = nomClasse.lastIndexOf(".") + 1;
            nomClasse = nomClasse.substring(debutNom);
            nomClasse = "vopimpl." + nomClasse.toLowerCase() + ".vue.Vue" + vue;

            Class nouvelleClasse = Class.forName(nomClasse);
            Constructor[] constructors = nouvelleClasse.getConstructors();
            VuePermission perm = new VuePermission(vue);
            Vue nouvelleVue = (Vue)constructors[0].newInstance(objBase, this, perm);
            vues.put(vue, nouvelleVue);
        }
        catch (Exception e){
            String message  = "Impossible d'attacher la vue " + vue + " ("
                            + e.getMessage() + ")";
            throw new VOPException(message);
        }
    }

    private void detacherVue(String vue) throws VOPException{
        try{
            vues.remove(vue);
        }
        catch (NullPointerException npe){
            String message  = "Impossible de détacher la vue " + vue + " ("
                            + npe.getMessage() + ")";
            throw new VOPException(message);
        }
    }

    final protected Vue getVue(String vue) throws VOPException {
        if (!vues.containsKey(vue))
            attacherVue(vue);
        Vue vueCourante = (Vue)vues.get(vue);
        VuePermission perm = vueCourante.getPermission();
        try{
            AccessController.checkPermission(perm);
        }
        catch(Exception e){
            String message  = "Accès interdit à la vue " + vue + " ("
                            + e.getMessage() + ")";
            throw new VOPException(message);
        }
        return vueCourante;
    }

    public Object execute(Subject subject, Method method, Object[] valeurs)
    throws VOPException{
        Object retour = Subject.doAsPrivileged(subject, new VOPPrivilegedAction(
                        method, valeurs, this), null);
        if (retour instanceof VOPException)
            throw (VOPException)retour;
        return retour;
    }
}
```

vop.app.VOPException

Identique à la classe de la solution de base

vop.securite.VOPCallBackHandler

```
public class VOPCallbackHandler implements CallbackHandler {

    public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof TextOutputCallback) {
                // display the message according to the specified type
                TextOutputCallback toc = (TextOutputCallback)callbacks[i];
                switch (toc.getMessageType()) {
                case TextOutputCallback.INFORMATION:
                    System.out.println(toc.getMessage());
                    break;
                case TextOutputCallback.ERROR:
                    System.out.println("ERROR: " + toc.getMessage());
                    break;
                case TextOutputCallback.WARNING:
                    System.out.println("WARNING: " + toc.getMessage());
                    break;
                default:
                    throw new IOException("Unsupported message type: " +
                    toc.getMessageType());
                }
            }
            else if (callbacks[i] instanceof NameCallback) {
                // prompt the user for a username
                NameCallback nc = (NameCallback)callbacks[i];
                System.err.print(nc.getPrompt());
                System.err.flush();
                nc.setName((new BufferedReader
                (new InputStreamReader(System.in))).readLine());
            }
            else if (callbacks[i] instanceof PasswordCallback) {
                // prompt the user for sensitive information
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                System.err.print(pc.getPrompt());
                System.err.flush();
                pc.setPassword(readPassword(System.in));
            }
            else {
                throw new UnsupportedCallbackException
                (callbacks[i], "Unrecognized Callback");
            }
        }
    }

    // Reads user password from given input stream.
    private char[] readPassword(InputStream in) throws IOException {
        char[] lineBuffer;
        char[] buf;
        int i;
        buf = lineBuffer = new char[128];
        int room = buf.length;
        int offset = 0;
        int c;

        loop:   while (true) {
                    switch (c = in.read()) {
                        case -1:
                        case '\n':
                            break loop;
                        case '\r':
                            int c2 = in.read();
```

```
                            if ((c2 != '\n') && (c2 != -1)) {
                                if (!(in instanceof PushbackInputStream))
                                    in = new PushbackInputStream(in);
                                ((PushbackInputStream)in).unread(c2);
                            }
                            else
                                break loop;
                        default:
                            if (--room < 0) {
                                buf = new char[offset + 128];
                                room = buf.length - offset - 1;
                                System.arraycopy(lineBuffer, 0, buf, 0, offset);
                                Arrays.fill(lineBuffer, ' ');
                                lineBuffer = buf;
                            }
                            buf[offset++] = (char) c;
                            break;
                    }
                }
                if (offset == 0)
                    return null;
                char[] ret = new char[offset];
                System.arraycopy(buf, 0, ret, 0, offset);
                Arrays.fill(buf, ' ');
                return ret;
            }
        }
```

vop.securite.VOPPrincipal

```
public class VOPPrincipal implements Principal, java.io.Serializable {

    private String name;

    public VOPPrincipal(String name) {
        if (name == null)
            throw new NullPointerException("illegal null input");
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return("VOPPrincipal:  " + name);
    }

    @Override
    public boolean equals(Object o) {
        if (o == null)
            return false;
        if (this == o)
            return true;
        if (!(o instanceof VOPPrincipal))
            return false;
        VOPPrincipal that = (VOPPrincipal)o;
        if (this.getName().equals(that.getName()))
            return true;
        return false;
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }
}
```

vop.securite.VOPPriviledAction

```
public class VOPPrivilegedAction implements PrivilegedAction {

    private Method objAPPMethod;
    private Object[] valeurs;
    private ObjApp objApp;

    public VOPPrivilegedAction(Method method, Object[] valeurs, ObjApp objApp){
        this.objAPPMethod = method;
        this.objApp = objApp;
        this.valeurs = valeurs;
    }

    public Object run(){
        try {
            return objAPPMethod.invoke(objApp, valeurs);
        }
        catch (Exception e) {
            String message = null;
            if ((Exception)e.getCause() instanceof VOPException)
                message  = "Probleme d'acces a la methode ("
                        +((Exception)e.getCause()).getMessage() + ")";
            else
                message  = "Probleme inconnu... (" + e.getMessage() + ")";
            return new VOPException(message);
        }
    }
}
```

vop.securite.VuePermission

```
public class VuePermission extends BasicPermission implements Serializable {

    public VuePermission(String perm){
        super(perm);
    }
}
```

vop.securite.login.ObjAppSecure

```
public abstract class ObjAppSecure {

    private ObjApp objApp;
    private Subject subject;

    public ObjAppSecure (){}

    // portée package
    void setObjApp(ObjApp objApp){
        this.objApp = objApp;
    }

    // portée package
    void setSubject(Subject subject){
        this.subject = subject;
    }

    protected Object execute(String methodName) throws VOPException{
        return execute(methodName, null, new Object[0]);
    }

    protected Object execute(String methodName, Class<?> params, Object[] valeurs) throws
VOPException{
        Method method = null;
        try{
            if (params == null)
                method = objApp.getClass().getDeclaredMethod(methodName);
            else
                method = objApp.getClass().getDeclaredMethod(methodName, params);
```

```
            return objApp.execute(subject, method, valeurs);
        }
        catch(Exception e){
            String message  = "Probleme avec la methode " + method
                            + " de l'object d'application (" + e.getMessage() + ")";
            throw new VOPException(message);
        }
    }
}
```

vop.securite.login.VOPLogin

```
public final class VOPLogin {

    private static VOPLogin vopLogin = new VOPLogin();

    private VOPLogin() {}

    public static ObjAppSecure login(String objBase) throws VOPException {
        Subject subject = getInstance().logUser();
        return getInstance().getObjAppSecure(objBase, subject);
    }

    // acces package... Attention!
    static VOPLogin getInstance(){
        return vopLogin;
    }

    boolean login(String nomUsager, char[] motPasse){
        // remplacer par un mécanisme de validation avec une BD, LDAP ou autre...
        // dans le cas d'une implémentation complète
        if ( (nomUsager.equals("admin") && motPasseValide(motPasse, "admin"))
           || (nomUsager.equals("invite") && motPasseValide(motPasse, "invite"))
           || (nomUsager.equals("archive") && motPasseValide(motPasse, "archive")))
            return true;
        return false;
    }

    private boolean motPasseValide (char[] motPasse, String motPasseValide){
        for (int x = 0; x < motPasse.length; x++)
            if (motPasse[x] != motPasseValide.charAt(x))
                return false;
        return true;
    }

    private ObjAppSecure getObjAppSecure(String objBase, Subject subject)
    throws VOPException{
        ObjApp objApp = getObjApp(objBase);
        ObjAppSecure objAppSecure = getObjAppSecure(objApp);
        objAppSecure.setObjApp(objApp);
        objAppSecure.setSubject(subject);
        return objAppSecure;
    }

    private ObjAppSecure getObjAppSecure(ObjApp objApp) throws VOPException{
        String className = objApp.getClass().getName() + "Secure";
        try{
            return (ObjAppSecure)Class.forName(className).newInstance();
        }
        catch(Exception e){
            String message  = "Impossible d'instancier l'object d'application secure "
                            + className + "(" + e.getMessage() + ")";
            throw new VOPException(message);
        }
    }

    private Subject logUser() throws VOPException {
        LoginContext loginContext = null;
        try {
            loginContext = new LoginContext("VOP", new VOPCallbackHandler());
        }
```

```
        catch (Exception e) {
            String message  = "Impossible de créer un contexte d'authentification ("
                            + e.getMessage() + ")";
            throw new VOPException(message);
        }
        try {
            loginContext.login();
            return loginContext.getSubject();
        }
        catch (LoginException le) {
            String message  = "L'authentification a échouée (" + le.getMessage() + ")";
            throw new VOPException(message);
        }
    }
    private ObjApp getObjApp(String objBase) throws VOPException{
        // A remplacer par quelque chose de mieux lors d'une implantation complète
        // Par exemple, enregistrer les ObjApp dans le RMIRegistry?
        if (objBase.compareTo("InfoEtudiant") == 0){
            InfoEtudiant infoEtudiant = new InfoEtudiant(
            "Dallaire Eric", -100, "maitrise");
            return new InfoEtudiantApp(infoEtudiant);
        }
        throw new VOPException("Objet d'application introuvable");
    }
}
```

vop.securite.login.VOPLoginModule

```
public class VOPLoginModule implements LoginModule {
    // État initial
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;
    // Le statut de l'authentification
    private boolean succeeded = false;
    private boolean commitSucceeded = false;
    // usager et mot de passe
    private String username;
    private char[] password;
    private VOPPrincipal userPrincipal;
    public void initialize(Subject subject, CallbackHandler callbackHandler,
    Map<java.lang.String, ?> sharedState, Map<java.lang.String, ?> options) {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;
    }
    public boolean login() throws LoginException {
        // prompt for a user name and password
        if (callbackHandler == null)
            throw new LoginException("Error: no CallbackHandler available " +
            "to garner authentication information from the user");
        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback("Entrez votre nom d'usager : ");
        callbacks[1] = new PasswordCallback("Entrez votre mot de passe : ", false);
        try {
            System.out.println("\nUsagers (mot de passe)");
            System.out.println("- admin (admin)");
            System.out.println("- archive (archive)");
            System.out.println("- invite (invite)\n");
            callbackHandler.handle(callbacks);
            username = ((NameCallback)callbacks[0]).getName();
            char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
```

```
            if (tmpPassword == null) {
                // treat a NULL password as an empty password
                tmpPassword = new char[0];
            }
            password = new char[tmpPassword.length];
            System.arraycopy(tmpPassword, 0, password, 0, tmpPassword.length);
            ((PasswordCallback)callbacks[1]).clearPassword();
        }
        catch (java.io.IOException ioe) {
            throw new LoginException(ioe.toString());
        }
        catch (UnsupportedCallbackException uce) {
            throw new LoginException("Error: " + uce.getCallback().toString() +
            " not available to garner authentication information " +
            "from the user");
        }
        try{
            succeeded = VOPLogin.getInstance().login(username, password);
        }
        catch(Exception e){
            succeeded = false;
        }
        if (succeeded)
            return true;
        else {
            username = null;
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
            throw new FailedLoginException("User Name or password Incorrect");
        }
    }

    public boolean commit() throws LoginException {
        if (succeeded == false)
            return false;
        else {
            // add a Principal (authenticated identity)
            // to the Subject
            // assume the user we authenticated is the VOPPrincipal
            userPrincipal = new VOPPrincipal(username);
            if (!subject.getPrincipals().contains(userPrincipal))
                subject.getPrincipals().add(userPrincipal);
            // in any case, clean out state
            username = null;
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
            commitSucceeded = true;
            return true;
        }
    }

    public boolean abort() throws LoginException {
        if (succeeded == false)
            return false;
        else if (succeeded == true && commitSucceeded == false) {
            // login succeeded but overall authentication failed
            succeeded = false;
            username = null;
            if (password != null) {
                for (int i = 0; i < password.length; i++)
                    password[i] = ' ';
                password = null;
            }
            userPrincipal = null;
        }
```

```
        else {
            // overall authentication succeeded and commit succeeded,
            // but someone else's commit failed
            logout();
        }
        return true;
    }

    public boolean logout() throws LoginException {
        subject.getPrincipals().remove(userPrincipal);
        succeeded = false;
        succeeded = commitSucceeded;
        username = null;
        if (password != null) {
            for (int i = 0; i < password.length; i++)
                password[i] = ' ';
            password = null;
        }
        userPrincipal = null;
        return true;
    }
}
```

vop.vue.Vue

```
public abstract class Vue {

    protected Object objBase;
    protected ObjApp objApp;
    private VuePermission perm;

    public Vue(Object monObjBase, ObjApp monObjApp, VuePermission perm){
        objBase = monObjBase;
        objApp = monObjApp;
        this.perm = perm;
    }

    public VuePermission getPermission(){
        return perm;
    }
}
```

vop.vue.pointvue.Administration

Identique à la classe de la solution de base

vop.vue.pointvue.Archive

Identique à la classe de la solution de base

vop.vue.pointvue.Finance

Identique à la classe de la solution de base

vop.vue.pointvue.PointVue

Identique à la classe de la solution de base

vopimpl.infoetudiant.InfoEtudiantApp

```
public class InfoEtudiantApp extends ObjApp {

    public InfoEtudiantApp(Object monObjBase){
        super(monObjBase);
    }

    private InfoEtudiant getObjBase(){
        return (InfoEtudiant)objBase;
```

```
    }
    private VueAdministration getVueAdministration() throws VOPException {
        return (VueAdministration)getVue("Administration");
    }
    private VueFinance getVueFinance() throws VOPException {
        return (VueFinance)getVue("Finance");
    }
    private VueArchive getVueArchive() throws VOPException {
        return (VueArchive)getVue("Archive");
    }
    public boolean estOK() throws VOPException {
        // La vue administrateur est obligatoire mais les autres vues sont optionnelles.
        boolean ok = true;
        ok &= getVueAdministration().estOK();
        try{
            ok &= getVueFinance().estOK();
        }
        catch (VOPException vope){
            System.out.println("OPTIONNEL : Acces interdit a la vue Finance");
        }
        try{
            ok &= getVueArchive().estOK();
        }
        catch (VOPException vope){
            System.out.println("OPTIONNEL : Acces interdit a la vue Archive");
        }
        return ok;
    }
    public void setNom(String nom) throws VOPException{
        // La vue administrateur est obligatoire
        getVueAdministration().setNom(nom);
    }
    public Vector getComposantes() throws VOPException{
        // La vue administrateur est obligatoire
        return getVueAdministration().getComposantes();
    }
    public boolean enDette() throws VOPException{
        // La vue administrateur est obligatoire
        return getVueAdministration().enDette();
    }
    public void setStatut(String statut) throws VOPException{
        getVueAdministration().setStatut(statut);
    }
    public void afficherComposantes() throws VOPException{
        // La vue archive est obligatoire
        getVueArchive().afficherComposantes();
    }
    public float getBalance() throws VOPException{
        /* Les vues finance et archive sont optionnelles
         * On compare les résultats des vues avec le résultat de ObjBase
         */
        float balance = getObjBase().getBalance();
        float balanceVue = balance;
        try{
            balanceVue = getVueFinance().getBalance();
        }
        catch (VOPException vope){
            System.out.println("OPTIONNEL : Acces interdit a la vue Finance");
        }
        if (balance != balanceVue)
            throw new VOPException("Impossible de balancer avec la vue Finance");
```

```
        try{
            balanceVue = getVueArchive().getBalance();
        }
        catch (VOPException vope){
            System.out.println("OPTIONNEL : Acces interdit a la vue Archive");
        }
        if (balance != balanceVue)
            throw new VOPException("Impossible de balancer avec la vue Archive");
        return balance;
    }
    public void paiement(float montant) throws VOPException{
        // La vue archive est obligatoire
        getVueFinance().paiement(montant);
    }
    public String getStatut(){
        return getObjBase().getStatut();
    }
    public String getNom(){
        // Aucune vue n'est nécessaire, accès direct à l'objet de base
        return getObjBase().getNom();
    }
}
```

vopimpl.infoetudiant.InfoEtudiantAppSecure

```
public class InfoEtudiantAppSecure extends ObjAppSecure{
    public boolean estOK() throws VOPException{
        return ((Boolean)execute("estOK")).booleanValue();
    }
    public void setNom(String nom) throws VOPException{
        Class<?> params = String.class;
        Object[] valeurs = {nom};
        execute("setNom", params, valeurs);
    }
    public String getNom() throws VOPException{
        return (String)execute("getNom");
    }
    public Vector getComposantes() throws VOPException{
        return (Vector)execute("getComposantes");
    }
    public boolean enDette() throws VOPException{
        return ((Boolean)execute("enDette")).booleanValue();
    }
    public void setStatut(String statut) throws VOPException{
        Class<?> params = String.class;
        Object[] valeurs = {statut};
        execute("setStatut", params, valeurs);
    }
    public void afficherComposantes() throws VOPException{
        execute("afficherComposantes");
    }
    public float getBalance() throws VOPException{
        Float returnObj = (Float)execute("getBalance");
        return returnObj.floatValue();
    }
    public void paiement(float montant) throws VOPException{
        Class<?> params = Float.class;
        Object[] valeurs = {Float.valueOf(montant)};
        execute("paiement", params, valeurs);
    }
```

```
    public String getStatut(){
        return null;
    }
}
```

vopimpl.infoetudiant.vue.VueAdministration

```
public class VueAdministration extends VueInfoEtudiant implements Administration{

    public VueAdministration (Object monObjBase, ObjApp monObjApp, VuePermission perm){
        super(monObjBase, monObjApp, perm);
    }

    public boolean estOK(){
        return !getComposantes().isEmpty();
    }

    public void setNom(String nom){
        getObjBase().setNom(nom);
    }

    public Vector getComposantes(){
        return getObjBase().getListeCours();
    }

    public boolean enDette(){
        return getObjBase().getBalance() < 0;
    }

    public void setStatut(String statut){
        getObjBase().setStatut(statut);
    }
}
```

vopimpl.infoetudiant.vue.VueArchive

```
public class VueArchive extends VueInfoEtudiant implements Archive {

    public VueArchive (Object monObjBase, ObjApp monObjApp, VuePermission perm){
        super(monObjBase, monObjApp, perm);
    }

    public void afficherComposantes(){
        Vector composantes = getObjBase().getListeCours();
        for (int x = 0; x < composantes.size(); x++){
            Cours cours = (Cours)composantes.get(x);
            System.out.println(cours.getNom());
            System.out.println(cours.getSigle());
            System.out.println(cours.getCote());
        }
    }

    public float getBalance(){
        return Math.abs(getObjBase().getBalance());
    }

    public boolean estOK(){
        return getObjBase().getStatut().compareTo("") != 0;
    }
}
```

vopimpl.infoetudiant.vue.VueFinance

```
public class VueFinance extends VueInfoEtudiant implements Finance {

    public VueFinance (Object monObjBase, ObjApp monObjApp, VuePermission perm){
```