

**I.M.A.G.**

**Maîtrise des Sciences et Techniques  
« Expert en Systèmes Informatiques »**

**Rapport de stage**

**Protection par capacité cachée  
sur carte à puce *Java Card***

**Sébastien Chassande-Barrioz / Stéphane Martin**

Juin - Août 1999

# REMERCIEMENTS

Nos remerciements les plus sincères vont à :

**Monsieur Roland Balter,**

Professeur à l'université Joseph Fourier de Grenoble, pour nous avoir accueilli au sein du projet SIRAC.

**Monsieur Daniel Hagimont ,**

Chargé de recherche à l'INRIA , qui fut un responsable de stage hors normes.

Et à toutes les autres personnes de l'équipe SIRAC pour leur accueil.

# Sommaire

<b>1</b>	<b>INTRODUCTION.....</b>	<b>4</b>
1.1	CONTEXTE DU STAGE .....	4
1.1.1	<i>INRIA Rhône-Alpes.....</i>	<i>4</i>
1.1.2	<i>Projet SIRAC.....</i>	<i>4</i>
1.2	RESUME DU PROBLEME .....	5
1.3	RESUME DU RESULTAT .....	5
<b>2</b>	<b>LA JAVACARD .....</b>	<b>6</b>
2.1	LA CARTE .....	6
2.1.1	<i>Historique .....</i>	<i>6</i>
2.1.2	<i>Caractéristiques techniques .....</i>	<i>6</i>
2.1.3	<i>Protocole de communication .....</i>	<i>8</i>
2.1.4	<i>Avantages de Java pour la programmation des cartes à puces.....</i>	<i>8</i>
2.2	APPLICATIONS VISEES .....	9
<b>3</b>	<b>LE MODELE DE PROTECTION.....</b>	<b>10</b>
3.1	UN MODELE A BASE DE CAPACITES .....	10
3.2	ECHANGES DE CAPACITES.....	11
3.3	EXEMPLE.....	13
3.4	REALISATION SUR JAVA .....	14
<b>4</b>	<b>APPLICATION À LA JAVA CARD .....</b>	<b>18</b>
4.1	MISE EN ŒUVRE DANS LA CARTE.....	18
4.2	MISE EN ŒUVRE LECTEUR / CARTE .....	18
4.2.1	<i>Restriction de Java Card sur le modèle .....</i>	<i>18</i>
4.2.2	<i>Implantation.....</i>	<i>19</i>
4.3	DESCRIPTION DE NOTRE OUTIL .....	19
4.3.1	<i>Interface graphique.....</i>	<i>19</i>
4.4	EXEMPLE.....	21
4.4.1	<i>Applet AirFrance .....</i>	<i>22</i>
4.4.2	<i>Applet Hertz.....</i>	<i>23</i>
4.5	LES OUTILS A NOTRE DISPOSITION .....	24
<b>5</b>	<b>CONCLUSION.....</b>	<b>25</b>

# 1 INTRODUCTION

## 1.1 Contexte du stage

Nous avons effectué notre stage à l'Institut National de Recherche en Informatique et en Automatique sous la tutelle de Daniel Hagimont. Nous avons travaillé au sein du projet SIRAC

### 1.1.1 INRIA Rhône-Alpes

L'INRIA Rhône-Alpes est la plus récente des cinq unités de recherche de l'INRIA, après celles de Rocquencourt, Rennes, Sophia-Antipolis et Nancy.

Cette Unité mène ses activités en étroite collaboration avec des laboratoires de recherche publics et privés, nationaux et internationaux installés dans la région. Elle entretient des rapports privilégiés avec l'institut d'Informatique et de Mathématiques Appliquées de Grenoble (IMAG).

Elle a son siège à Montbonnot en Isère, dans un bâtiment qui accueille aujourd'hui 15 projets de recherche, comprenant des chercheurs des Universités (UJF et INPG), ainsi que des ingénieurs de la société Bull.

### 1.1.2 Projet SIRAC

SIRAC est un projet commun entre l'Institut National de Recherche en Informatique et Automatique (INRIA), l'Institut National Polytechnique de Grenoble (INPG) et l'Université Joseph Fourier.

L'évolution des systèmes informatiques vers les applications réparties est un phénomène en pleine expansion qui touche tous les secteurs d'activité : communication et prise de décision de l'entreprise, gestion industrielle et financière, publication et diffusion de l'information, santé, transports, loisirs. Elle est favorisée par plusieurs facteurs. Nous observons d'une part une croissance significative des capacités réseaux de communication et d'autre part, une tendance des entreprises à décentraliser leurs structures de production et de décision.

Les outils et services disponibles aujourd'hui sur les plates-formes d'usage courant sont insuffisantes pour maîtriser la complexité due au développement et au déploiement de grandes applications réparties. Les systèmes pour le développement des ces applications doivent fournir l'infrastructure nécessaire à l'exécution répartie de programmes et au partage d'informations entre les machines d'un réseau mais aussi des outils d'aide à la construction, à l'installation et à l'administration des applications. Ces deux aspects définissent le champ d'action du projet SIRAC, dont l'objectif général est de concevoir et réaliser un environnement pour le développement et l'exécution d'applications réparties.

Son champ d'activité couvre les environnements de développement d'applications, le support système pour la construction de serveurs d'informations et les protocoles et services pour communications mobiles.

## 1.2 Résumé du problème

Aujourd'hui les cartes à puce font partie de notre vie courante. Les plus courantes sont les cartes bancaires. La puce ne contient que des données relatives à l'utilisateur (comme son code) et des protocoles de communications avec les lecteurs dont la majeure composante est la sécurité (vérification du code).

Avec *Java Card*, il devient possible d'installer sur la carte des petits programmes sous forme d'*applets* Java écrites dans une version allégée du langage.

Ainsi, la carte peut dorénavant contenir, en plus des données personnelles de l'utilisateur, des programmes appartenant à plusieurs entités différentes. Ces entités peuvent être des organismes commerciaux ou publics, des sociétés diverses en partenariat ou en concurrence. Nous pouvons donc nous attendre à une réduction sensible du nombre de cartes dans notre quotidien, une seule carte pouvant faire office à la fois de carte bancaire, de carte de parking et de badge d'accès !

Il devient alors nécessaire de fournir à ces sociétés des garanties sur la sécurité des informations présentes sur la carte, non seulement lors des accès depuis un lecteur (suivant une sorte de RPC) mais aussi entre ces *applets* qui peuvent (depuis la version 2.1 de Java Card de Juin 99) partager des objets.

Le modèle de protection par capacités cachées développé, au sein du projet SIRAC de L'INRIA Rhône-Alpes, s'applique parfaitement au problème.

Le but de ce document est de montrer l'application du modèle à capacités cachées pour la protection d'*applets* s'exécutant sur une carte à puce.

## 1.3 Résumé du résultat

Durant les trois mois de notre stage, nous avons développé un prototype visant à valider le concept de protection par capacités cachées (conçu par Daniel Hagimont) sur une carte à puce. Le modèle à capacité caché appliqué à la carte à puce va faire l'objet d'un dépôt de brevet en partenariat entre Gemplus (l'un des principaux fabricant de carte à puce au monde) et l'INRIA.

Le prototype produit a été présenté en démonstration le 24/08/1999 à Jean-Jacques Vandewalle, ingénieur de recherche chez Gemplus. Il inclut :

- La protection par capacités cachées entre applets dans la carte.
- Un générateur de stubs gérant les RPC entre le lecteur de carte et la carte.
- La protection par capacités cachées entre le lecteur et la carte, intégré au générateur de stubs.

Dans notre implémentation, nous avons simplifié au maximum la programmation de l'application du coté lecteur de carte (domaine *Out Card*), ainsi que la programmation de l'*applet* en elle même (*In Card*).

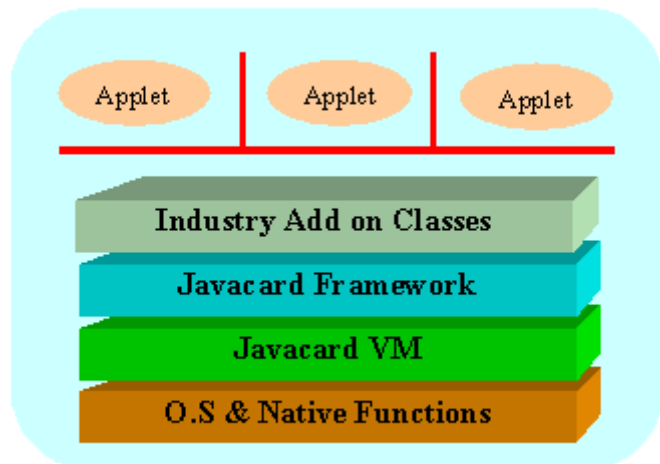
Nous avons démontré l'intérêt du modèle à capacité cachées à travers la réalisation d'une application pilote.

# 2 LA JAVACARD

## 2.1 La carte

### 2.1.1 Historique

En mars 1996, un petit groupe d'ingénieurs de Schlumberger au Texas ont souhaité simplifier la programmation des cartes à puces tout en préservant la sécurité des données. Presque immédiatement, ils ont analysé que ce problème avait déjà été posé par le chargement de code sur le World Wide Web. La solution avait été Java. Malheureusement vu la taille de la mémoire disponible sur une carte à puce seulement un sous-ensemble de Java pouvait être utilisé (la taille de la JVM (Java Virtual Machine) et du système de runtime ne devant pas dépasser 12 kø). C'est ainsi qu'est né Java Card, le premier langage à objets adapté aux cartes à puces. Schlumberger et Gemplus sont les co-fondateurs du Java Card Forum qui recommande des spécifications à JavaSoft (la division de Sun à qui appartient Java Card) en vue d'obtenir une standardisation du langage et il s'occupe aussi de la promotion des APIs (Application Programming Interface) Java Card pour qu'il devienne la plate-forme standard de développement des applications pour les smart cards.



L'architecture d'un système JavaCard

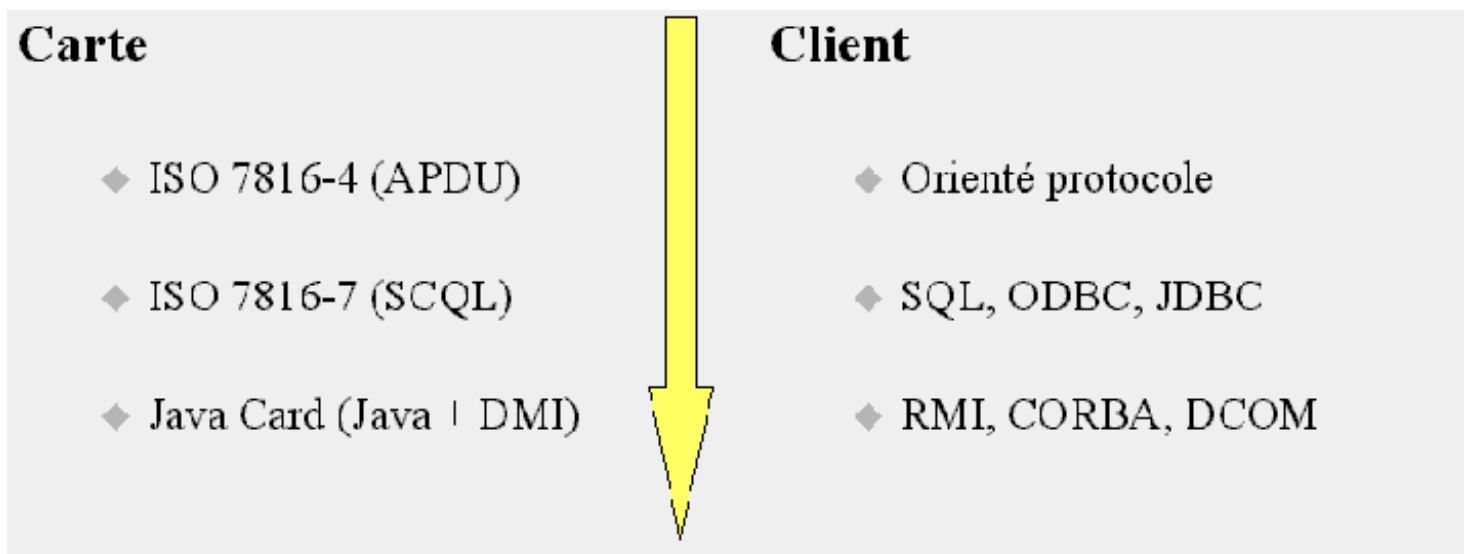
### 2.1.2 Caractéristiques techniques

Il y a deux sortes de cartes à puce : la carte à mémoire servant seulement au stockage d'information utilisant la logique pour accéder à l'information (ex : une carte de téléphone avec 50 crédits) et la carte à puce intelligente (celle qui nous intéresse dans ce rapport) qui est un mini-ordinateur. Elle contient un micro-contrôleur 8 bits (les plus connus étant les composants 6805 de Motorola et le 8051 d'Intel) qui a des propriétés de sécurité.

Sa mémoire est composée de RAM (Random Access Memory : mémoire volatile à accès rapide, jusqu'à 2 Ko) où sont stockées les données temporaires (i.e. ces données sont perdues lors de la perte de courant), d'EEPROM (Electrical Erasable Programmable ROM : mémoire non-volatile réinscriptible, jusqu'à 32 Ko) où sont stockées les données sur le possesseur de la carte (ex : le numéro de compte bancaire, la clé privée d'encryption, etc.) et de la ROM (Read-Only Memory : mémoire non-volatile à lecture seule (jusqu'à 64 Ko) où résident les programmes de la carte).

## Quelques précisions

- L'OS de la carte. L'OS de la carte est gravé en ROM au moment de la fabrication. Il contient une machine virtuelle Java simplifiée. Le développement du code des *applets* nécessite des compétences carte. Ce code est généralement développé par les fabricants.
- Programmation des lecteurs. Il existe un protocole standard de communication entre le système hôte et le lecteur. La communication se fait par tableaux de *byte*, les informations codées en hexadécimal. Les drivers des lecteurs offrent uniquement une API de transport des messages entre le lecteur et la carte.
- Evolutions à venir. Pour l'instant, le protocole de communication avec la carte ne permet pas de faire un appel de procédure vers la carte complet (RPC). Dans ce RPC, la carte n'est pour l'instant que Serveur (*RPC Simplex*). Ainsi, du côté lecteur, il n'est pas possible de passer en paramètre d'appel d'une méthode de la carte une référence à un objet local au lecteur. Ce problème résolu, la carte pourra devenir un composant à part entière d'applications réparties et être intégrée comme support dans n'importe quel système d'information.



## Evolution des possibilités d'utilisation d'une carte à puce

### 2.1.3 Protocole de communication

Le protocole ISO 7816 définit un ensemble de standards internationaux pour les cartes à circuit intégré avec contacts que doivent suivre les programmeurs d'application pour que les opérations soient possibles sur n'importe quel terminal (le lecteur de carte ou CAD : Card Acceptance Device) dans le monde entier qui fournit le pouvoir électrique à la carte. La structure de communication utilisée entre le CAD et la carte est un format standardisé de donnée : une APDU (Application Protocol Data Unit) qui est un message de commande du CAD vers la carte ou un message de réponse de la carte vers le CAD (figure 1 : Format des APDUs).

APDU de Commande							APDU de Réponse			Obligatoire
cla	ins	p1	p2	lc	data	le	data	Sw1	Sw2	Facultatif

#### Format des APDUs

*cla* indique la classe de la commande, *ins* le code de l'instruction à faire, *p1* et *p2* sont les paramètres de la commande, *lc* est la longueur en octet des données, *le* est le nombre maximum des octets attendus pour la données de l'APDU de réponse. *Sw1* et *sw2* indiquent comment s'est passée la commande (le mot d'état).

Les APDU sont interprétées lorsqu'elles sont reçues sur la carte.

### 2.1.4 Avantages de Java pour la programmation des cartes à puces

- C'est un langage haut niveau, à objets permettant l'encapsulation des données et la réutilisation de code. Ainsi les programmeurs n'auront plus besoin d'apprendre les langages assembleur 6805 ou 8051 pour réaliser une application pour une carte. La mise au point de programmes sera plus rapide et les temps de développement seront beaucoup plus courts. De plus, c'est un langage sûr : pas de manipulation de pointeurs, pas d'allocation de mémoire, ... Cette caractéristique limite grandement les erreurs de programmation.
- Il est exécutable sur n'importe quelle carte à puce car il est exécuté sur une machine virtuelle. Le code de l'application est compilé en byte-code qui est exécuté par la machine virtuelle Java (JVM). Donc cette portabilité permettra d'écrire du code qui fonctionnera sur n'importe quel micro-processeur de cartes à puces (« Write Once, Run Anywhere »).
- Il possède un modèle de sécurité qui permet à plusieurs applications de coexister en sécurité sur la même carte.
- Le gros avantage de Java Card est la possibilité de charger dynamiquement, n'importe quand, une nouvelle applet sur la carte. Cela signifie que le code des applets pourra être mis à jour.

## 2.2 Applications visées

Les applications visées pour la Java Card sont toutes celles où des informations doivent être échangées et/ou traitées de manière sécurisée.

Pour valider notre prototype, nous avons imaginé un scénario à base d'un système de points de fidélité. Ce scénario est grandement inspiré du système *Fréquence Plus* que la société *Air France* propose à ses clients. Dans le programme *Fréquence Plus* se trouve un ensemble de partenaires, notamment *Hertz* (société de location de voitures) et *Air France*. Lorsqu'un client loue une voiture ou achète un billet d'avion, il cumule des points de fidélité. Avec suffisamment de points de fidélité, un client peut obtenir un billet d'avion gratuit. Pour gérer les points de fidélité, trois applets sont installées dans une carte, une qui représente *Fréquence Plus*, une pour *Air France* et une pour *Hertz*.

Un client possédant une Java Card arrive dans l'agence de location de voiture (*Hertz*). La location d'un certain type de véhicule, pour une durée déterminée et un certain nombre de kilomètres va entraîner pour le client un gain de points *Fréquence Plus*. Pour prendre en compte ces nouveaux points, l'applet *Hertz* appelle une méthode de l'applet *Fréquence Plus* et lui passe en paramètre un objet qui représente la location de la voiture (contenant notamment le nombre de kilomètre effectués). Le client va ensuite chez *Air France* où il lui est possible, suivant son nombre de points courant, de payer un billet d'avion en points *Fréquence Plus*.

Du point de vue de la protection, les applets de *Fréquence Plus* et de *Hertz* sont mutuellement méfiantes. L'applet *Hertz* a seulement le droit d'annoncer une nouvelle location de voiture à *Fréquence Plus* (et pas de créditer un nombre arbitraire de points). L'applet *Fréquence Plus* a seulement le droit de lire l'objet représentant la location de voiture (et pas de le modifier).

La sécurité est assurée par le modèle à capacités cachées (décrit dans la partie suivante).

# 3 LE MODELE DE PROTECTION

## 3.1 Un Modèle à Base de Capacités

Le modèle de protection choisi est basé sur des capacités logicielles. Le modèle à capacité a l'avantage de faciliter l'échange de droits d'accès à l'exécution, ce qui est un de nos objectifs (dynamisme).

Une capacité est un jeton qui identifie un objet et qui définit des droits d'accès sur cet objet, c'est à dire le sous ensemble des méthodes qui peuvent être appelées sur cet objet. Pour appeler un objet A, un objet B doit posséder une capacité sur l'objet A autorisant cet appel. Lorsqu'un objet est créé, une capacité est retournée au créateur et contient les droits maximaux sur l'objet créé. Cette capacité peut être utilisée pour accéder à l'objet, mais elle peut également être copiée et transmise à un autre objet. Lorsqu'une capacité est copiée, ses droits peuvent être restreints afin de limiter les droits accordés à un autre objet.

Chaque objet s'exécute donc dans un environnement de protection dans lequel il a accès à tous les objets qui lui appartiennent. Le transfert de capacité d'un objet à un autre est réalisé lors d'un appel de méthode. Lorsqu'une référence d'objet est passée en paramètre, ce passage de référence peut s'accompagner d'un passage de capacité.

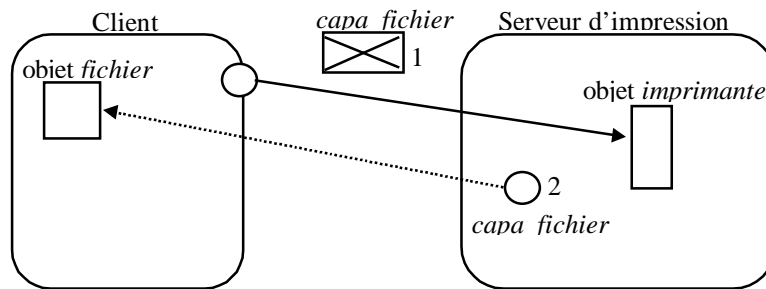


Figure 1. Serveur d'impression

Afin d'illustrer ce modèle à capacités, considérons l'exemple d'une application serveur d'impression qui exporte un objet *imprimante* permettant d'imprimer un fichier (Figure 1).

Une capacité sur l'objet *imprimante* est donnée aux objets clients, leur permettant d'imprimer un fichier. Lorsqu'un client veut imprimer un fichier, l'objet *imprimante* doit recevoir le droit de lire le fichier à imprimer. Le client passe donc, lors de l'appel (1), une capacité en lecture sur le fichier (*capa\_fichier*) à l'application appelée. Cette capacité permet à l'objet *imprimante* de lire le contenu du fichier (2).

### 3.2 Echanges de Capacités

Comme on l'a vu dans la section précédente, l'emploi de capacités logicielles donne un modèle où les droits peuvent être échangés dynamiquement entre objets. Il reste alors à définir comment un programmeur d'objet va spécifier les règles d'échange de droits avec les autres objets.

L'intérêt pour le modèle de protection est la modularité. En effet, ce modèle ne fournit pas des extensions au langage de programmation, permettant de spécifier le passage de capacités en paramètre lorsqu'une méthode est appelée sur un objet appartenant à un autre objet. Le modèle permet d'exprimer les transferts de capacités entre objets à l'aide d'un langage de définition d'interface (IDL). Etant donné qu'une interface peut être décrite indépendamment du code qui la réalise, décrire la politique d'échange des droits au niveau de l'interface permet de séparer clairement l'expression de la protection du code de l'application.

Ainsi, un IDL a été étendu pour permettre de spécifier les capacités qui doivent accompagner le passage de référence en paramètre dans un appel de méthode. Cet IDL permet de définir des *vues*.

Une vue est une interface incluant la définition d'une politique de protection. Une vue est associée à une capacité et décrit :

- les méthodes qui sont autorisées par cette capacité,
- les échanges de capacités lorsqu'il y a passage de référence en paramètre dans une des méthodes de la vue. Les capacités à passer sont décrites en termes de vues.

La définition des vues est naturellement récursive, car une vue décrit les capacités qui accompagnent certains paramètres, cette description se faisant en terme de vues. Pour ce faire, un nom est associé à chaque vue lors de sa déclaration.

Dans l'exemple du serveur d'impression décrit auparavant, deux vues peuvent être définies pour la classe *File* : une vue *reader* qui n'autorise que la méthode *read* et une vue *writer* qui autorise les deux méthodes *read* et *write*. Pour la classe *Printer*, on définit la vue *user* qui autorise la méthode *print* dont la signature est la suivante : `void print ( reader File f )`. Cette signature indique que lorsqu'une référence à un fichier est passée en paramètre de la méthode *print*, une capacité avec la vue *reader* doit être transmise à l'application appelé.

Le modèle pourrait s'en tenir à une définition de la protection du côté de l'appelé si nous considérons une architecture client/serveur où la protection vise essentiellement à protéger le serveur contre ses clients. Cependant, il faut prendre en compte la suspicion mutuelle entre les objets. Chaque objet doit pouvoir contrôler les capacités qu'il exporte aux autres objets (l'appelant comme l'appelé).

De plus, nous voulons assurer l'autonomie des objets. Plus précisément, il n'est pas possible pour un programmeur d'objet de vérifier la politique de protection d'un objet offrant un service (même si cette définition est enregistrée dans un serveur officiel) lors de la programmation de l'objet, car à cet instant, le programmeur ne sait pas encore forcément quels objets il va rencontrer.

Pour ces deux raisons, chaque objet va définir sa propre vision de la politique de protection à mettre en œuvre lorsqu'il interagit avec d'autres objets. A une capacité sont donc associées deux vues, la vue de l'objet appelant et la vue de l'objet appelé.

La vue de l'objet appelé décrit :

- Les méthodes autorisées.
- Pour chaque paramètre à l'aller (référence  $R$  reçue par l'appelé), la vue décrit les capacités qui seront données par l'objet appelé lorsque la référence  $R$  est utilisée pour un appel de méthode. Cette vue décrit donc, du point de vue de l'appelé, les capacités que l'objet accepte d'exporter.
- Pour chaque paramètre de retour (référence  $R$  donnée par l'appelé), la vue décrit la capacité retournée avec la référence  $R$ .

et la vue de l'objet appelant décrit :

- Pour chaque paramètre à l'aller (référence  $R$  donnée par l'appelant), la vue décrit la capacité donnée avec la référence  $R$ .
- Pour chaque paramètre de retour (référence  $R$  reçue par l'appelant), la vue décrit les capacités qui seront données par l'objet appelant lorsque la référence  $R$  est utilisée pour un appel de méthode. Cette vue décrit donc, du point de vue de l'appelant, les capacités que l'objet accepte d'exporter.

Ce schéma symétrique est la réponse aux problèmes de la suspicion mutuelle et de l'autonomie des objets. Chaque objet (l'appelant et l'appelé) définit les capacités qu'il accepte d'exporter.

Chaque objet définit les vues correspondant aux interfaces Java qu'il utilise. Ces vues sont prises en compte comme suit. Lorsqu'une application exporte une référence à un objet en utilisant le serveur de nom, il spécifie la vue associée à cette référence, donc la capacité qu'il désire exporter. De cette façon, l'objet définit également toutes les capacités qui seront exportées suite à des appels de méthode en utilisant la capacité exportée dans le serveur de nom (ces exportations sont spécifiées par la définition des vues de l'objet).

Lorsqu'un objet récupère une capacité dans le serveur de nom, il définit la vue qu'il désire associer à cette capacité (de son côté). De cette façon, il définit également toutes les capacités qui seront exportées suite à des appels de méthode en utilisant la capacité récupérée dans le serveur de nom.

Toute capacité échangée entre ces deux objets prendra en compte les vues des deux objets.

### 3.3 Exemple

Afin d'illustrer le schéma d'expression du modèle de protection, nous reprenons l'exemple du serveur d'impression (légèrement modifié) et donnons ci-dessous les interfaces Java mises en jeu et les vues qui sont définies par l'objet client et par l'objet serveur d'impression.

```
interface Printer_itf {
    void init ();                // initialiser l'imprimante
    Job_itf run (Text_itf text); // envoyer un texte à imprimer
}
interface Text_itf {
    String read();              // lire le texte
    void write (String s);      // écrire le texte
}
interface Job_itf {
    void stop ();               // arrêter l'impression
}
```

Ces interfaces sont les interfaces Java qui sont mises en commun entre les objets pour pouvoir coopérer. Pour mettre son service à disposition des clients, le serveur d'impression exporte dans le serveur de nom une instance de sa classe *Printer*. Cette classe est une réalisation de l'interface *Printer\_itf*. De son côté, l'application cliente récupère cette instance et peut appeler une méthode (*init* ou *run*) sur cette instance grâce à l'interface *Printer\_itf* (cette interface Java est utilisée pour forcer le type de l'instance reçue du serveur de nom).

Lorsqu'un fichier doit être imprimé, le client appelle la méthode *run* en passant une référence sur l'objet fichier à imprimer qui est de classe *Text*. Cette classe est une réalisation de l'interface *Text\_itf*. La méthode *run* retourne une référence à une instance de la classe *Job* qui est une réalisation de l'interface *Job\_itf*. Le client peut appeler la méthode *stop* sur cette instance pour arrêter l'impression s'il le désire.

La spécification de la protection pour ces deux objets vise à éviter les problèmes suivants :

- le serveur d'impression ne veut pas que le client puisse appeler la méthode *init* et réinitialiser l'imprimante,
- le client ne veut pas que l'imprimante puisse appeler *write* sur le texte et le modifier.

Dans notre schéma de protection, le client et le serveur définissent alors les vues suivantes :

#### client

```
view client implements Printer_itf {
    void init ();
    Job_itf run (Text_itf text pass reader);
}
view reader implements Text_itf {
    String read();
    void not write (String s);
}
```

#### serveur d'impression

```
view server implements Printer_itf {
    void not init ();
    Job_itf run (Text_itf text);
}
```

Chaque objet définit un ensemble de vues définissant sa politique de protection. Chaque vue « implémente » l'interface Java correspondant au type des objets qu'elle protège. Le mot clé **not** est placé devant une définition de méthode qui est interdite dans une vue. Lorsqu'une référence est passé en paramètre dans une vue, le programmeur peut spécifier la vue à passer avec le mot clé **pass**. Si aucune vue n'est spécifiée, cela signifie qu'aucune restriction n'est opéré pour ce passage de paramètre.

Dans cet exemple, le serveur d'impression définit la vue *server* qui interdit aux clients les appels à la méthode *init*. Aucune restriction n'est faite sur les paramètres à l'aller et au retour de la méthode *run*. Le client déclare la vue *client* qui spécifie que, lorsqu'une référence sur un texte est passée à la méthode *run*, la vue *reader* doit être passée. La vue *reader* interdit la méthode *write* au serveur d'impression. Notons que le client n'a pas de raison de s'interdire la méthode *init* ; c'est une décision du serveur d'impression.

Lorsque le serveur d'impression enregistre une instance de la classe *Printer* dans le serveur de nom, il y associe sa vue *server*. Lorsque le client récupère cette référence du serveur de nom, il y associe sa vue *client*. Ces deux vues ainsi que celles qu'elles utilisent (*reader*) servent au contrôle des échanges des droits entre les objets. En résumé, chaque objet spécifie sa politique de protection de façon autonome et modulaire, sous la forme de vues qui sont des interfaces incluant la définition des échanges de capacités entre les objets.

Le modèle de protection décrit dans la section précédente à été réalisé sur Java. Dans cette section, nous décrivons la réalisation choisi par Mr Hagimont.

### **3.4 Réalisation sur Java**

Pour l'implantation de ce modèle de protection, il est utilisé le fait que les références à des objets Java sont presque des capacités. En effet, étant donné que Java est un langage sûr, il n'est pas possible dans un programme Java de forger une référence à un objet et d'appeler une méthode sur cet objet. Ceci implique que si un objet O1 crée un objet O2, l'objet O2 n'est pas accessible à partir des autres objets de l'environnement Java, tant que O1 ne passe pas explicitement une référence vers O2 à ces autres objets. Ce passage ne peut se faire que par passage de paramètre lorsqu'un objet appelle O1 ou lorsque O1 appelle un autre objet. Ainsi, tant qu'une application ne passe pas des références à ses objets, ceux-ci sont protégés contre les autres objets. Une référence Java peut donc être vue comme une capacité, mais ce n'est

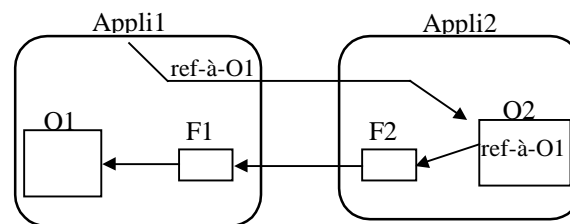
pas réellement une capacité, puisqu'il n'est pas possible de restreindre les méthodes pouvant être appelées sur cette référence. Il est ajouté un mécanisme inspiré de la notion de Proxy permettant de restreindre les droits associés à une référence Java.

L'implantation repose sur la notion d'objets *filters* insérés entre l'appelant et l'appelé. Pour chaque vue définie par un objet, une classe filtre est générée et une instance de cette classe est insérée pour protéger l'objet.

Lorsqu'une référence d'objet est transmise à un autre objet (en paramètre d'un appel de méthode), nous passons à la place de la référence sur l'objet une référence sur une instance de la classe filtre, générée à partir de la vue spécifiée par l'application qui donne la référence.

Cette classe filtre implante toutes les méthodes définies dans l'interface que la vue réalise. Elle déclare une variable d'instance qui pointe sur la référence de l'objet réellement passé en paramètre et qui est utilisée pour retransmettre les appels de méthode autorisés vers le bon objet. Si une méthode non autorisée est appelée sur l'instance filtre, une exception est déclenchée.

Cette instance filtre, qui remplace la référence passée en paramètre d'un appel de méthode, est insérée par l'objet appelant ou plus précisément par l'instance filtre utilisée pour l'appel de méthode. Ce sont donc les objets filtres qui, lorsqu'ils sont appelés, sont chargés d'installer des objets filtres pour les références passées en paramètre.



**Figure 2.** Gestion des objets filtres.

Réciproquement, lorsqu'une référence est reçue par un objet, une instance filtre est passée à la place, dont la classe a été générée à partir de la vue spécifiée par l'objet qui reçoit la référence.

Ainsi, deux objets filtres sont insérés entre l'appelant et l'appelé comme cela est représenté sur la figure 2.

Supposons qu'une référence à un objet *O1* est passée en paramètre d'un appel à un objet *O2* entre *Appli1* et *Appli2*, que ce soit un paramètre à l'aller ou au retour. Les objets filtres installés entre *Appli1* (l'appelant) et *O2* (l'appelé) vont alors insérer les objets filtres suivants entre *Appli2* et *O1* :

- *F1* : l'objet filtre correspondant à la vue spécifiée par *Appli1* pour le paramètre *O1*. Ce filtre assure que seules des méthodes autorisées peuvent être appelées par *Appli2* et il insère des filtres pour le compte de *Appli1* pour les paramètres des appels à *O1*.
- *F2* : l'objet filtre correspondant à la vue spécifiée par *Appli2* pour le paramètre *O1*. Ce filtre insère des filtres pour le compte de *Appli2* pour les paramètres des appels à *O1*.

Voici ce que donne notre réalisation sur l'exemple du serveur d'impression. Les classes filtres générées sont données ci-dessous.

### client

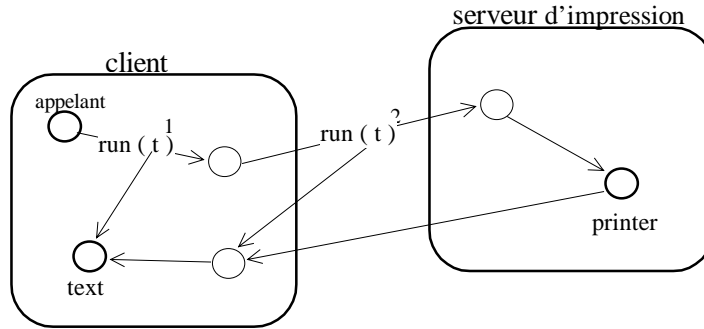
```
public class reader implements Text_itf {
    Text_itf obj;
    public reader(Text_itf o) {
        obj = o;
    }
    public String read() {
        return obj.read();
    }
    public void write(String s) {
        Exception !!!
    }
}
```

```
public class client implements Printer_itf {
    Printer_itf obj;
    public Printer_stub(Printer_itf o) {
        obj = o;
    }
    public void init() {
        obj.init();
    }
    public Job_itf run(Text_itf text) {
        reader stub = new reader(text);
        return obj.run(stub);
    }
}
```

### serveur d'impression

```
public class server implements Printer_itf {
    Printer_itf obj;
    public Printer_stub(Printer_itf o) {
        obj = o;
    }
    public void init() {
        Exception !!!
    }
    public Job_itf run(Text_itf text) {
        return obj.run(text);
    }
}
```

La classe *reader* du client retransmet les appels à *read* vers l'instance réelle, mais elle ne laisse pas passer les appels à *write*. De même, la classe *server* du serveur d'impression ne laisse passer que les appels à *run*. Dans la classe *client* du client, la méthode *run* prend en paramètre un objet *text* pour lequel une capacité avec la vue *reader* doit être passée. Pour ce paramètre, la méthode *run* de la classe filtre crée une instance de la classe filtre *reader* et l'initialise avec le vrai paramètre, puis retransmet l'appel en passant l'objet filtre créé à la place du vrai paramètre.



**Figure 3.** Objets filtre dans le serveur d'impression.

La figure 3 illustre l'insertion des objets filtres dans le cas du serveur d'impression. A l'étape (1), l'appel de la méthode *run* transmet la référence vers le vrai objet *texte*. Cet appel est réalisé sur l'objet filtre du client pour la référence à l'objet *printer*. Cet objet filtre crée un objet filtre pour le paramètre *texte* et transmet l'appel à l'objet filtre du serveur d'impression (2) en passant l'objet filtre du paramètre. L'appel est ensuite transmis au vrai objet *printer*. Ultérieurement, l'appel de la méthode *read* sur l'objet *texte* passe à travers l'objet filtre du *texte* installé par le client.

Notons que si le client avait défini une vue *vue\_job* pour la référence à l'instance de la classe *Job* retournée par la méthode *run*, la classe filtre *client* aurait alors la méthode *run* suivante :

```

public Job_itf run(Text_itf text) {
    reader stub = new reader(text);
    return new vue_job(obj.run(stub));
}
  
```

Cette méthode crée à la fois un objet filtre pour le paramètre (*Text*) à l'aller et pour le paramètre (*Job*) au retour. Dans le cas du serveur d'impression, la vue *vue\_job* n'est pas nécessaire.

L'exemple du serveur d'impression que nous avons utilisé jusqu'ici ne visait qu'à expliquer le modèle et son implantation sur un exemple très simple.

# 4 APPLICATION A LA JAVA CARD

## 4.1 Mise en œuvre dans la carte

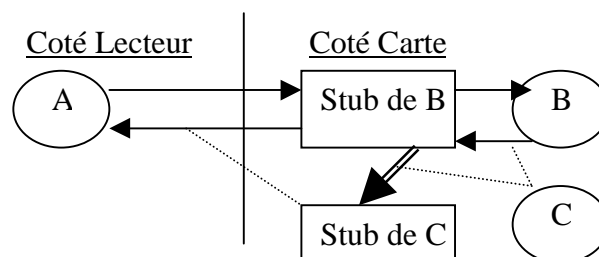
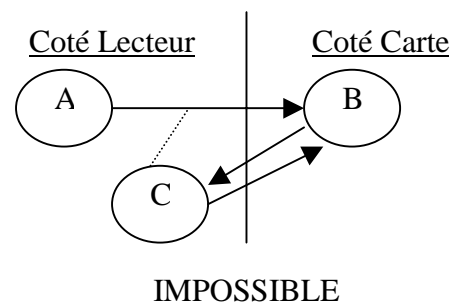
Le principe des appels entre deux objets reste le même. La politique d'accès à un objet est toujours définie par ses objets filtres (que nous appelons *stubs*) qu'il choisit d'exporter.

Nous avons cependant du intégrer ce modèle dans l'environnement de la *Java Card* avec de forte contraintes de programmation.

## 4.2 Mise en œuvre Lecteur / Carte

### 4.2.1 Restriction de *Java Card* sur le modèle

Supposons que nous sommes dans le cas où deux objets A et C sont du côté lecteur et un objet B est du côté Carte (cf figure à droite). Du fait que *Java Card 2.1* n'autorise pas les appels de méthodes entre la carte et le lecteur, il est impossible pour B d'appeler A ou C. Donc A ne peut appeler B, avec en paramètre une référence à C. En effet, la référence transmise en paramètre est inutilisable, puisque la carte ne peut y accéder. On peut donc déduire, en regardant attentivement le modèle à capacité cachées, que les *stubs* du côté lecteur sont inutiles. En effet, ceux-ci servent uniquement à protéger les références passées en paramètres.



L'application du modèle n'est plus symétrique.

En revanche, un objet A du côté lecteur peut, lors d'un appel à un objet B coté carte recevoir en retour d'appel de la méthode une référence. L'objet A peut ainsi acquérir de nouvelles références (voir schéma ci-dessus).

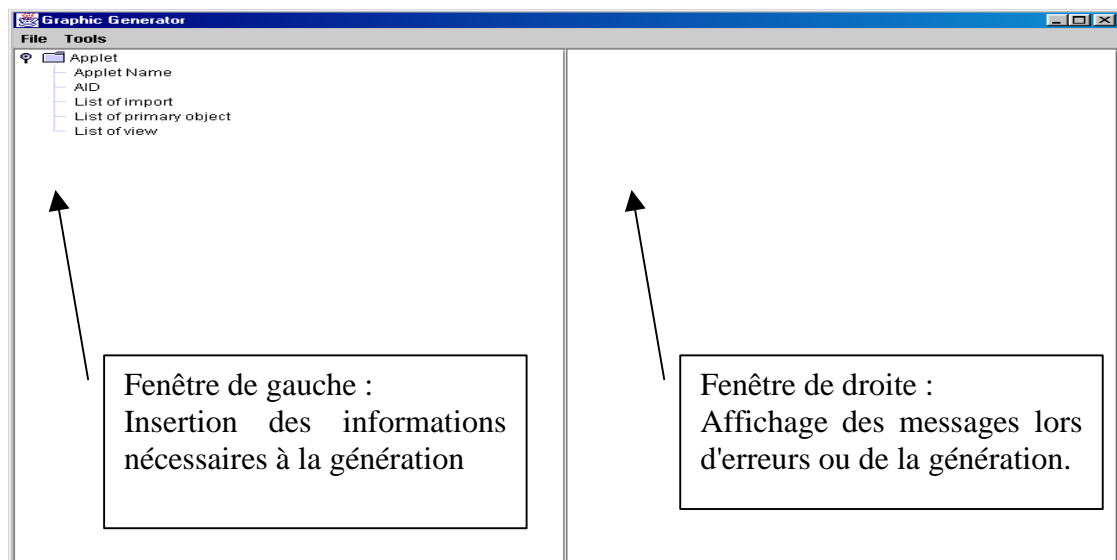
## 4.2.2 Implantation

Nous avons utilisé le *simulateur Java Card* de Sun qui nous a été fourni, car la dernière version de la carte – *Java Card 2.1* – n’était pas encore disponible. Cette version ne comprend pas encore de RPC entre le lecteur et la carte. Nous avons dû en réaliser un à partir des primitives d’échange d’APDU, puis nous avons intégré à ce RPC le modèle de protection par capacité pour qu’il permette les appels protégés et les échanges de capacités entre le lecteur et la carte.

## 4.3 Description de notre outil

L’outil que nous avons réalisé durant le stage, permet de générer une *applet Java Card* ainsi que son module de communication côté lecteur. Lorsqu’on désire mettre un objet (*Java Card*) sur la carte, il faut le mettre dans une *applet*. Le générateur permet de faire ceci mais aussi de définir une politique de protection à base de capacités cachées. L’objet est protégé contre tous les autres objets. Ainsi les autres *applets* situées sur la carte et les programmes situés du côté lecteur ne peuvent accéder à l’objet que par la politique de protection (capacité) définie au moment de la génération de l’*applet*. On peut mettre plusieurs objets dans une *applet* et définir les accès de tous les objets.

### 4.3.1 Interface graphique



**Graphic Generator**  
File Tools

- Applet
  - Applet Name
  - AID
  - List of import
  - List of primary object
  - List of view

Callout boxes:

- Champ éditable où on tape l'AID de l'applet, un nombre hexadécimal.
- Utiliser le click droit pour insérer un nouvel objet primaire.
- Champ éditable ou on insère le nom de l'applet, une chaîne de caractère.
- Utiliser le click droit pour insérer un nouvel import.
- Utiliser le click droit pour insérer une nouvelle vue

**Graphic Generator**  
File Tools

- Applet
  - AHertz
    - 1111111111111111
    - List of import
      - ItfRepository.\*
    - List of primary object
      - Hertz
        - Hertz\_itf
        - Capacity
      - List of view
        - HertzAll
          - Hertz\_itf
            - List Of Method
              - annuler
                - void
                - Authorized
                - List Of Parameter
                  - short
              - clore
                - void
                - Authorized
                - List Of Parameter
                  - short
                  - short
              - reserver
                - short
                - Authorized
                - List Of Parameter
                  - short
                  - short

Callout boxes:

- Champ d'un import inséré. Le champ est éditable.
- Objets primaires insérés :  
*Nom* : éditable, chaîne de caractère  
*Interface* : éditable, chaîne de caractère  
*Capacité* : Faire un click droit pour ajouter une capacité correspondant à l'interface. Il faut que des vues soient définies
- Nom de la vue
- Nom de l'interface que représente la vue.
- Type du retour de la méthode. On peut le change si c'est une référence.
- Champ permettant d'autoriser ou d'interdire l'appel d'une méthode dans cette vue
- Paramètres de la méthode. On peut les change si ce sont des références.

#### 4.4 Exemple

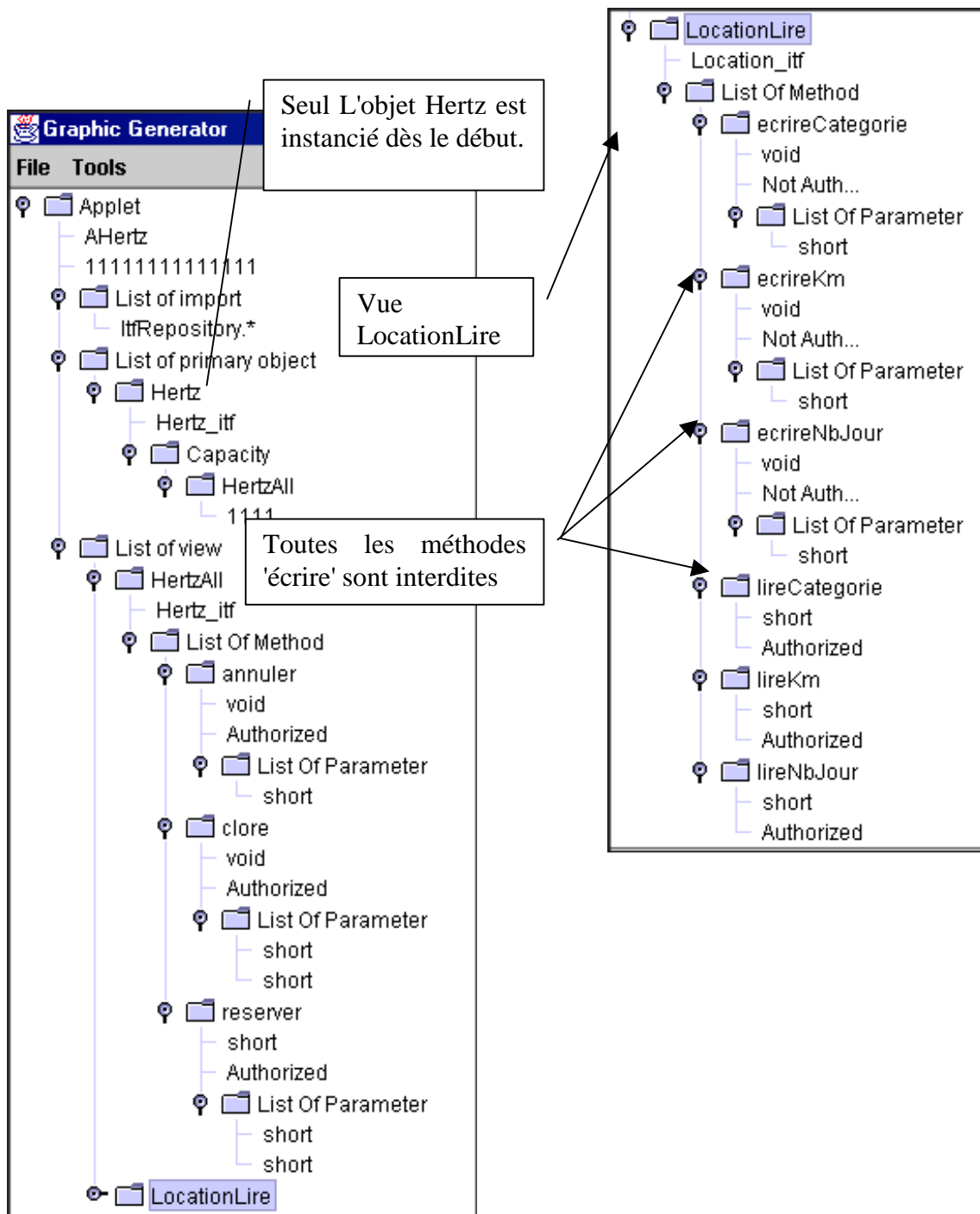
Reprenons l'exemple des deux *applets* *Hertz* et *Air France*.

Voici les interfaces des objets communicant :

| Interface de AirFrance                                                                                                                                                                                                                                                                                                              | Interface de Hertz                                                                                                                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> public interface AirFrance_itf { // la création de l'objet AirFrance crée l'objet FQ+ public short reserver (     short classeVol,     short numVol,     short date ); public void annuler( short numReservation ); public void payer(     short numReservation,     boolean avecPoint ); public short lirePoints(); } </pre> | <pre> Public interface Hertz_itf { Public short reserver (     short categorie,     short nbJour );  public void annuler( short numLocation);  public void clore(     short numLocation,     short nbKm ); } </pre>                                                      |
| Interface de FrequencePlus                                                                                                                                                                                                                                                                                                          | Interface de Location                                                                                                                                                                                                                                                    |
| <pre> public interface FrequencePlus_itf {  public void ajouter( short nbPoints );  public short lire();  public void utiliser( short nbPoints );  public void ajouterLocation(     <b>Location_itf</b> location ); } </pre>                                                                                                        | <pre> public interface Location_itf {  public void ecrireNbJour( short nbJour );  public short lireNbJour ();  public void ecrireKm ( short km );  public short lireKm ();  public short lireCategorie ();  public void ecrireCategorie (     short categorie); } </pre> |



#### 4.4.2 Applet Hertz



#### **4.5 Les outils à notre disposition**

Pour valider cette réalisation, nous avons utilisé le *simulateur Java Card* de Sun, car la dernière version de la carte – *Java Card 2.1* – n'était pas encore disponible. Malheureusement, nous avons perdu énormément de temps à faire fonctionner ce simulateur. En effet, nous avons dû modifier sa première couche de traitement qui ne permettait pas de faire des appels 'dynamiques' (il n'acceptait qu'un script d'appel de méthodes prédéfini à l'avance).

Le portage sur les Java Card 2.1, qui seront disponibles en septembre, ne posera aucun problème. Il simplifie une partie du code que nous avons produit.

# 5 CONCLUSION

La possibilité de charger des programmes Java sur une carte à puce modifie notre vision banale d'une carte, qui n'est plus seulement une simple mémoire morte mais devient un véritable mini-ordinateur. Ces programmes devant s'échanger des données, il faut proposer un modèle de protection évolué à la fois simple (espace mémoire limité) et performant (données sensibles).

Nous avons travaillé sur l'implémentation du modèle à capacités cachées pour la protection d'*applet Java Card* s'exécutant sur une même carte. Nous avons réalisé un prototype pour valider le concept et présenter le modèle à la société Gemplus, dans le but de réaliser un transfert de technologie et intégrer le concept dans la future version de leur kit de développement *Java Card* : GEMEXPRESSO .

Les perspectives apportées par la miniaturisation des processeurs et l'évolution des systèmes répartis nous amènent à posséder, dans notre poche, un outil contenant l'intégralité des informations dont nous avons besoin quotidiennement.

Pouvons nous réellement imaginer le contenu exhaustif de notre portefeuille préféré réduit à une simple carte ?